CD INCLUDED

PROGRAMMING THE
NINTENDO
GAME BOY
ADVANCE:
THE UNOFFICIAL GUIDE

Jonathan S. Harbour

Programming The Nintendo Game Boy Advance:
The Unofficial Guide

Programming The Nintendo Game Boy Advance: The Unofficial Guide

Jonathan S. Harbour

Project Editor: Estelle Manticas

Copy Editor: Laura Gabler

Technical Reviewer: André LaMothe

Programming Reviewer: Emanuel Schleussinger

Software Reviewer: Peter Schraut

For selfless support and love, for accepting

the inequity of my preoccupation with games

and fully supporting it, this book is dedicated

to my wonderful wife,

Jennifer Rebecca Harbour

Acknowledgements

I am thankful for the encouragement of my wife, Jennifer, who has been wonderfully supportive of my writing career. I love our life together, and our kids, Jeremiah and Kayleigh, bring me more joy than I thought possible. I also owe a thank you to two pair of friends who have greatly encouraged me on a personal level: Justin and Kim Galloway, and Milford "Wade" and Lindsay Eutsey. I thank God for the blessing of your friendship. I am grateful for a supportive family: my parents, Ed & Vicki; my sister, Joy; my niece, April; in-laws, Dave & Barb Yoder; David; Steve, Andrea, and Stephen; Jason, Autumn, and Tater; and Barbara Jean. On another level, I thank the Lord for giving me the drive to do this sort of work, which is both enjoyable and insanely difficult at the same time. I am thankful for having learned many life lessons from Focus On The Family and Dr. James Dobson, as well as from the extraordinary *Left Behind* series by Tim LaHaye and Jerry Jenkins.

This was perhaps the most enjoyable book I have written, not solely due to the content (which is utterly compelling!), but also due to two events that took place while this book was being written. The first experience was during the Game Developer's Conference 2003, where I met several editors from Premier Press face-to-face for the first time. After the Game Developer's Choice Awards, presented by the International Game Developer's Association (IGDA), Premier Press held a private dinner at a nice restaurant in downtown San Jose, which was a fun meet-and-greet with André LaMothe, Heather Hurley, Mitzi Koontz, Heather Talbot, Todd Jensen, as well as other fellow authors. Thanks to all of you for doing such a great job with this series; I have enjoyed working with all of you. The second experience that made this book memorable was a celebration a few days later on the launch of *DarkBASIC* with fellow author Joshua Smith, our families, and my two favorite freelancers, Cathleen Snyer and Estelle Manticas. Thank you for doing such great work, it's been a great pleasure working with you.

I would like to thank one of the greatest game designers of all time, John Romero, for not only being such a great encouragement and role model for aspiring game developers, but also for agreeing to write the foreword. As Jacopo says, "I'm your man!" Thanks to my good friend, Joshua R. Smith, for loaning me the Flash Advance Linker.

Greets go out to two guys who were invaluable while writing this book, for they provided up-to-the-minute updates to the development tools. Emanuel Schleussinger, for the excellent GBA SDK used in this book, called *HAM*, and to Peter Schraut for the excellent IDE

called *Visual HAM*. These tools saved me literally hundreds of hours of work. Not only did they provide superior development tools to me and the GBA community at no charge, these two were invaluable as proofreaders of the manuscript. Thank you both for getting me out of many dead ends and helping resolve coding problems along the way. This book reflects your efforts.

A book of any size and on any subject involves much work even after a manuscript is done. It takes a while just to read through a programming book once, so you can imagine how much is involved in reading through it several times, making changes and notes along the way, refining, correcting, perfecting. I am indebted to the hard work of the editors, artists, layout specialists who do such a fine job. Really, as far as total hours go, the author's work is only perhaps 40% (or less) of the total time spent on getting a book into print. Thank you especially to Laura Gabler for copy editing the manuscript, to André LaMothe for his technical expertise, and to Estelle Manticas for managing the project.

About The Author

Jonathan S. Harbour has been programming games for 15 years, first with Microsoft GW-BASIC and Turbo Pascal, then on to Turbo C, Borland C++, Watcom C++, and 80386 assembler. After finally bridging the gap to Windows programming, he has spent time with Borland Delphi, Visual Basic, Visual C++, and more recently, Visual Studio .NET. Jonathan graduated from DeVry Institute of Technology in 1997 with a B.S. degree in Computer Information Systems, and has since worked for cellular, aerospace, pharmaceutical, education, medical research, healthcare, and game companies. In his spare time, Jonathan enjoys spending time with his family, reading fiction, playing console video games, watching movies, and working on his classic Mustangs.

Contents

Part III
Meditating On The Hardware

Part IV
The Mother of All Appendixes

Foreword

For over 10 years, programming the Game Boy has been a mystery. Unless you are an approved Nintendo developer, the only way to learn how to develop on the Game Boy family has been to piece together a million scraps of information off the internet, play around with secret register addresses you heard whispered somewhere, and basically just shove lots of crazy values in the register lists you've pieced together to see just what would happen.

But now, that era has passed. Jonathan Harbour has tackled the huge task of compiling all this esoteric and cryptic information into the most useful Game Boy programming book you'll find. Even Nintendo's documentation doesn't read this good (and we're Approved Nintendo Developers, so we know) and the example code and explanations are now understandable to English-reading lifeforms.

You'll find out soon that programming the Game Boy is really not that difficult if you're familiar with C programming and using pointers. We designed our game, *Hyperspace Delivery Boy!*, originally for the Pocket PC computer. When we ported the game to the Game Boy Advance, we could still retain our original cross-platform code architecture and just rewrite the areas of the game that dealt with graphics, sound, and file access (since it's now all just memory in ROM). It's always a great idea to write your game for your target platform and take advantage of its strengths, but coding the Game Boy Advance doesn't totally require you to do that and still create a successful game.

What are you waiting for? Turn the page and get ready to make your own game on a cartridge!

John Romero

Game Designer

http://www.monkeystone.com

Introduction

The Game Boy Advance is a sophisticated handheld video game machine, with ▪ powerful 32-bit microprocessor, 16-bit graphics, stereo digital sound, and yet small enough to fit in your pocket. Millions of Game Boy Advance units have been sold already, with upwards of 100,000 units a week shipping around the world during peak months, making this the highest selling video game system in history, with over 140 million units sold in all. The affordable price of the Game Boys, large library of games, relatively low price of games and accessories, plus portability make this a very compelling video game system. Investing in development personnel and tools is more likely on a prolific system like Game Boy Advance, which has a long life span (like earlier Game Boy models). 14 years, 4 models, and 600 games later (at the time of this writing), Game Boy is the most successful video game franchise in history, outselling all others by a wide margin. Adding icing to the cake, the latest Game Boy fully supports all the game cartridges from the earlier models!

Game publishing companies and development studios like the Game Boy for its relatively low cost of entry and high potential margins. While a Game Boy cartridge might cost more to produce than a CD-ROM or DVD game disc, there are many factors that amount to strong profits nonetheless. One factor is that Game Boy is almost universally considered an accessory device, not a primary video game system. As such, families with one or more children will often have a console (such as the Nintendo GameCube) in the living room, as well as a Game Boy. Families with two or more children are also likely to buy a Game Boy for each child. Although the demographic for Game Boy users is primarily young children, ▪ large percentage of Game Boy owners are young adults (18-24) and older.

Game Boy Development

Until recently, it was nearly impossible for a hobby or student programmer to get involved in the console market. The problem is this: in order to get hired, console game developers require some level of experience with the hardware; however, one can't gain that experience without first being hired. The key word here is *hardware*. Anyone can gain experience writing games for a PC, because the development machine is equivalent to the end-user's machine. But this is not the case with consoles such as the Game Boy Advance, which requires a custom hardware interface, special development tools (the compiler, linker, emulator, link cables, etc). These tools are very expensive, and require a special license with the video game manufacturer (Nintendo, in this case), which also requires a

non-disclosure agreement (NDA). This agreement says that you are not to share the secrets of the console or development tools with anyone who is not an officially licensed developer. In other words, console games are developed safely behind closed doors.

One solution for the aspiring programmer was usually to find employment with a game developer as a game tester (also called QA, or quality assurance person), and then gradually learn the ropes by working with the programmers while testing games, with the hope of being promoted to a development position. Another possibility is to find employment with a PC game studio, and gain experience working on PC games before moving on to consoles. Experienced developers are usually easy to train on console hardware, so they are often considered for employment. The PC market is much easier to break into than the console market. Either of these solutions requires patience, determination, drive, and passion for writing games and learning the tricks of the trade, and no one is guaranteed to succeed. Most find enough satisfaction out of hobby programming that they are content in that respect. But the video game industry (like the computer game industry) is growing by leaps and bounds, so the need for highly motivated and talented programmers is greater today than at any time before.

So You Want To Be A Console Programmer?

There is now a solution to the circular problem of finding a job as a console game developer. The Game Boy Advance uses a popular microprocessor (the ARM7 CPU) for which there are software tools available, because this chip is used in hundreds of consumer electronics products: VCRs, DVD players, TVs, satellite receivers, MP3 players, DSL and cable modems, Pocket PCs, GPS transceivers, in addition to the Sega Dreamcast and Nintendo Game Boy Advance. In case you are wondering, the Dreamcast featured a powerful 3D chip in addition to its ARM CPU, and the Dreamcast CPU and Game Boy Advance CPU are similar (though the Dreamcast is faster).

The result is that public domain assemblers and compilers have been written for ARM processors, and the ARM Corporation itself has released tools into the public domain for use with these products. These tools were quickly adapted for Game Boy Advance, and talented programmers were creating emulators and writing Game Boy Advance programs before the handheld was even officially released by Nintendo. (The same was true for the Dreamcast as well).

Software Development Kits

You are probably wondering at this point: how does one write Game Boy programs, let alone compile them and run them on an actual Game Boy? The development tools have matured in the two years (at the time of this writing) since the Game Boy Advance was released. There are now numerous public domain (or freeware) compilers, assemblers, and emulators vying for market share in the Game Boy development community.

The software development kit I selected for this book is a complete integrated development environment (IDE) for writing, compiling, and running Game Boy Advance programs, right on the Windows or Linux desktop. That package is called *HAM*, and was developed by Emanuel Schleussinger. The IDE is called *Visual HAM*, and was created by Peter Schraut. In order to test the programs, HAM comes with VisualBoyAdvance, a top-notch emulation program that runs compiled Game Boy Advance ROM images in Windows (or Linux). That's it! HAM is all you need to learn the art and science of programming the Game Boy Advance, and this wonderful development tool package is free.

Although there are tools available for Linux, I have not included any information in this book about using the Linux version of HAM. Since the same Game Boy Advance code will compile under Windows or Linux, you may wish to download the Linux version of HAM from Emanuel's Web site at http://www.ngine.de. The same ROM images that you will compile in each chapter of this book will run just as well in Linux as in Windows, because the same emulator is available for Linux.

Now, what happens when you have written a really cool game or demo and want to run it on a real Game Boy Advance unit? Well, there are basically two options. First, there is a multi-boot cable device that will download and run compiled ROM binaries on the Game Boy Advance, using the link cable port. Second, there is a more elaborate option. You can write to your own flash cartridge. This might sound difficult and expensive, but it is neither. A flash linker is a small device that plugs into your PC's parallel or USB port and reads/writes the flash cartridge. The process is similar to using a reader for SmartMedia, CompactFlash, and MMC/SD cards, with which you might be familiar if you have a digital camera. Once written, you can plug the flash cartridge into the Game Boy Advance just like any other game and power it up.

Obviously this is not a quick process, and requires a minute or two at least, and adds wear and tear to the Game Boy Advance and the cartridge over time. While it is fun to show your games and demos to friends, the multi-link cable is much faster, because it doesn't require

a cartridge. In fact, you leave the cartridge slot on the Game Boy Advance empty, plug in the multi-link cable, power up the Game Boy Advance, and the compiled program is downloaded and run directly in the little machine. Visual HAM and the HAM development kit comes with the emulator and the various download programs, ready to go. You are literally able to compile and run your own code on your Game Boy Advance. The multi-link cable and flash linker are covered in Chapter 4, "Starting With The Basics," along with details on where you can get one. One of the most popular Web sites for ordering these hardware items, as well as aftermarket Game Boy Advance accessories, is http://www.lik-sang.com.

Is This Book For You?

My goal with this book is to get you a job as a Game Boy Advance programmer. Although you may be a hobby programmer, or perhaps a professional video game developer already, my basis and assumption for this book is that you are an aspiring game programmer. The result is that I don't waste any time talking about subjects that won't help you in that respect. This is a very focused book that stays close to the subject matter, and as a result it is not as large as some books. However, most game programming books try to cover as much as possible. Those are books about writing PC games. This is a console programming book!

Some things are similar between consoles and PCs, but consoles are at a far lower level than PCs when it comes to game development. For one thing, at least on the Windows platform, all commercial games use the DirectX library, a monstrously large runtime for interfacing with computer hardware. In a sense, DirectX turns the hordes of different PCs into a single console platform. While console programmers are guaranteed that the machine will run exactly the same for every person, that is not the case with PCs. DirectX helps to homogenize all the PCs so programmers don't go insane trying to support all the different brands, with different CPUs, graphics chips, and so on (which is how it used to be back in the MS-DOS era).

You will need to be proficient in the C language in order to follow along in this book. Remember the assumption! If you want to be a game programmer, you must know C already. If you don't know C at all, you will definitely need a primer before getting into the later chapters of this book. There are many beginner books on C available, such as *C Programming for the Absolute Beginner* by Michael Vine. You needn't even buy a recent book on C, because Game Boy Advance programs follow the ANSI C standard, which has been around for decades. For example, one of my textbooks in college was *C: An Introduction to Programming* by Jim Keogh, et al, and I used this book as a reference while

writing Game Boy Advance code. C is an easy language to learn, but very difficult to master: that is part of the mystery of this language, and the source of it's widespread use. C is ▪ very low-level language that is only a few degrees above assembly language.

I don't cover C++ because it is overkill for Game Boy Advance development. Most of the important aspects of programming the Game Boy Advance (such as sprites) are handled by the built-in hardware routines, and don't require extensive programmer intervention. For example, a sprite blitter is a given on the Game Boy Advance, so you will definitely not have to write one yourself! It's built into the hardware (which means, that functionality is provided by the manufacturer in the design of the system). However, I'm not going to argue the pros and cons, because I personally love the C++ language. C is simply easier to understand while learning the basics of Game Boy Advance programming. If you wish to use C++ yourself, you may do so, because the HAM toolkit does support C++.

This book does not cover C++, but you may still use C++ for writing Game Boy programs if you like, since the HAM SDK includes a C++ compiler.

System Requirements

The development tools required to write Game Boy Advance programs have very low system requirements, and will probably run just fine on any Windows-based PC. Even the lowliest old Pentium 133 will probably work, as long as DirectX is installed (for the emulator). However, here is a realistic minimum PC:

* Windows 95, NT, or later
* Pentium II 300 MHz
* 128 MB memory
* 200 MB hard drive space
* 8 MB standard video card

Book Summary

This book is divided into four parts:

Part I: The Zen of Getting Started

This first section of the book includes the introductory information you will need to get started in Game Boy Advance development. Included is an overview of the console industry,

the Game Boy Advance hardware, and how to install and use the HAM SDK (including Visual HAM and the VisualBoyAdvance emulator).

Part II: Being One With The Pixel

This section is dedicated to getting pixels on the Game Boy Advance screen, including chapters that cover all six video modes (three bitmap-based, and three tile-based), coverage of sprites, and also working with backgrounds, including special effects like panning, rotating, zooming, and alpha blending.

Part III: Meditating On The Hardware

This section focuses on the hardware aspects of the Game Boy Advance other than the video system, such as using interrupts and timers to retain a consistent frame rate, converting and playing sound files, checking for button presses, and using low-level assembly language to speed up code.

Part IV: The Mother of All Appendixes

This final section of the book includes the appendices, such as the ASCII chart, book and Web site listing, a Game Boy Advance hardware reference, answers to the chapter quizzes, and instructions for using the CD-ROM.

Part I

The Zen of

Getting Started

Welcome to Part I of *Programming The Nintendo Game Boy Advance: The Unofficial Guide*. Part I includes four chapters that introduce you to the Game Boy Advance. Starting with an overview of the video game industry (and a history of Nintendo that leads up to the Game Boy), this part goes into detail about the internal workings of the Game Boy Advance, and then provides a tour of the integrated development environment, Visual HAM, the HAM development toolkit, and Visual Boy Advance emulator. Finally, this part shows how to write, compile, and run several Game Boy programs, using the emulator, and also on an actual Game Boy.

Chapter 1 — Welcome To The Console World

Chapter 2 — Game Boy Architecture In A Nutshell

Chapter 3 — Game Boy Development Tools

Chapter 4 — Starting With The Basics

Chapter 1

Welcome To The Console World

Welcome to the new horizon of console programming. This may be the first book of its kind to cover the intricacies and complexity of programming ■ video game console. The Game Boy Advance is ■ proprietary handheld video game system and is one of the many wonderful and enjoyable consoles developed by Nintendo. The Game Boy Advance is ■ successor of the original Nintendo Entertainment System (NES) and Super NES and has reestablished some of the greatest game franchises of all time (which were lost when Nintendo delved into the 3D realm with the Nintendo 64 and GameCube).

This chapter is a historical overview of the industry, the company, and the machine. While not directly helpful in a programming sense, this material is nonetheless necessary in order to properly understand this little machine. By understanding the Game Boy's target audience, history, strengths, weaknesses, features, and flaws, one is better able to maximize the potential of the games that will be written for this handheld console. This chapter also delves into the console industry, describing where the Game Boy fits in with other consoles, and basically sets the pace for the rest of the book.

Here are the highlights of this chapter:

- A general introduction to video games
- A brief history of Nintendo
- A note about the importance of having goals

Portable Video Games!

Take a look at Figure 1.1 for a moment. This is a very small video game system called the Nintendo Game Boy Advance. As you are reading this book, I assume you know this already. But take a second look. The Game Boy Advance (also called GBA) is so small, so unassuming! What is all the fuss about regarding this little machine? Well, for starters, it's a self-contained video game console in a handheld case. What is packed away inside that case . . . now *that* is what has everyone so excited. The GBA is what many have called the *perfect* console. The notable exception to this endorsement is a dim color screen that is difficult to see. However, there is a solution to that problem, as I will discuss in the next chapter. A *truly* perfect console is a GBA with a backlit screen (and I'm not talking about the SP model). But I don't want to give away all the details right off the bat . . . .

Figure 1.1
The Game Boy Advance, circa 2001.

Getting into the Nintendo Thing

Possibly the greatest and most well-known video game character in history is Mario, who first appeared in the arcade game Donkey Kong in 1981. Donkey Kong was the first game designed by Shigeru Miyamoto, and it was hugely popular in the United States. Although originally named Jumpman, and originally a carpenter, the likeness of Mario was first established in this seminal video game character. Jumpman the carpenter was transformed into Mario later in Donkey Kong Jr. and then gained a new profession as a plumber in Mario Bros., oddly enough.

The 2D side-scrolling classic game that put the NES at the top of the industry, *Super Mario Bros.*, was revived for Game Boy Color in an accurate conversion called *Super Mario Bros. Deluxe*. In that respect, as the first Game Boy with a color screen, the Game Boy Color is a portable version of the NES. Many great old favorites were converted and updated from

NES, such as *Super Mario Bros.*, *R-Type*, *The Legend of Zelda* (*Link's Awakening*, *Oracle of Ages*, and *Oracle of Seasons*), *Castlevania*, and literally hundreds more.

The Game Boy Advance is a whole new banana, equivalent to a souped-up Super NES, with ▪ landscape-oriented 4:3 ratio LCD screen, a fast CPU, and plenty of memory. (I cover the specifics of the hardware in the next chapter.) The Game Boy Advance continues the tradition set forth by the Game Boy Color and expands upon some great game franchises, as well as offering new remakes of classic games.

Console Contemplation

If you are a PC guru, you will have to put your prejudices on the shelf when working with consoles, because hardware is not directly comparable. For one thing, computers are multipurpose machines. Yes, a significant number are dedicated to gaming, but consoles are specifically built to do one thing: play video games. As such, consoles are extremely efficient at displaying graphics. The GameCube, for example, is equipped with 48 MB of RAM. But that is not how console programmers think. In the console realm, everything is a bit. Bytes are the rule of the PC realm because originally everything on the PC revolved around characters: data entry, text screens, Teletype terminals, dot-matrix printers, and so on.

A character on a PC is based on the ASCII code, wherein each character is 1 byte. A console, in contrast, doesn't even have a text mode. Consoles work exclusively with bits, and as you know, there are 8 bits in 1 byte. So, when talking about consoles such as the GameCube, you should properly speak the terminology: It has 384 megabits (or Mbits) of memory. This will make so much more sense when we go over the Game Boy Advance hardware in detail in the next chapter.

Definition of a Video Game

Okay, this might go without saying, but it is important to get the terminology down before getting too deep into programming a console, especially if your background is in PCs. So what is ▪ video game? *The American Heritage Dictionary* defines a video game as "an electronic or computerized game played by manipulating images on a video display or television screen." In contrast, a computer game is defined as "a game played against a computer."

35

Programming The Nintendo Game Boy Advance: The Unofficial Guide
Copyright (c)2003 by Jonathan S. Harbour -- http://www.jharbour.com

See the distinction? Video games and computer games are not one and the same, just as PCs are not in the same category as consoles. Since we humans find comfort with categorizations, let's get one thing clear up front: Max Payne would get owned by Sam Fisher. Categorize *that* in your list of facts, and bring on the bullet time! In fact, rumor says Payne came over to the consoles to get some respect, while Fisher went to the PC to do some more damage.

A blockbuster video game is more than the sum of its bits. There is a mysterious aura around a cartridge that brings so much gaming goodness. Word to that game! A master like Shigeru Miyamoto or Yu Suzuki comes along about once every decade and presents ■ genre-busting ubergame that fosters about a hundred copycat games and keeps the gaming industry rocking until the next genius comes along—or until one of the masters produces ■ sequel or something new.

A Brief History of Nintendo

Nintendo did not always make video games. In fact, the name *Nintendo* was established in 1951, and the company was actually founded way back in 1889. That company was called Marufuku Company, founded in Japan by Fusajiro Yamauchi. The important factor to consider is that Marufuku was involved in games, even if just card games. The card games that Marufuku manufactured were called Hanafuda playing cards, and in 1902 Yamauchi expanded into other types of playing cards.

In 1951 Marufuku Company was renamed Nintendo Playing Card Company. The word *Nintendo* consists of three kanji characters that translate to "leave luck to heaven." Nintendo got into the toy manufacturing business in the 1970s, building toys such as light guns. Nintendo's first video game machine was actually a license to sell the Magnavox Odyssey in Japan in 1975. Nintendo's experience with the Odyssey helped the company to develop its own video games. In 1977, in a joint venture with Mitsubishi, Nintendo created the TV-Game ▌ and TV-Game 15 systems, which were Pong copycats.

In 1978 Nintendo built its first arcade game, a small table-sized cabinet called *Computer Othello*, which featured 10 buttons. This game was not a serious competitor for *Space Invaders*, which also came out in 1978, but did help to launch Nintendo's arcade game division. The year 1980 saw the introduction of *Radar Scope*, which sold poorly (leaving Nintendo with 2,000 unsold cabinets), but an interesting lead-in story. Keep in mind that, at

this point, the Atari 2600 dominated the home console market, while Nintendo was busy gaining experience in the arcade business.

Shigeru Miyamoto

The president of Nintendo, Hiroshi Yamauchi, decided to replace *Radar Scope* boards with a new game rather than recall the unsold cabinets, but that new game would have to be able to run on the *Radar Scope* boards. Enter Shigeru Miyamoto. Yamauchi hired Miyamoto as a staff artist, as a personal favor for a friend, Miyamoto's father. Yamauchi tasked Miyamoto with the job of writing a game for the *Radar Scope* boards, so a simple upgrade could be performed to the existing cabinets.

Miyamoto had no idea how to write a game, so he worked with the video game designers to translate his designs into a game. His design called for a small animated carpenter that could run, jump, climb ladders, and defeat a gorilla named Donkey Kong in order to save a blonde girl (who was held captive by Donkey Kong at the top of the screen). *Donkey Kong* firmly established Nintendo as a force to be reckoned with in the arcade business, and the company went on to produce many more arcade games (as shown earlier).

Miyamoto was finally credited formally by Nintendo in 1996 (at the launch of Super Mario 64) for his contribution to the company and the game industry.

Home Consoles

In 1983 Nintendo introduced the 8-bit NES in Japan, with launch titles *Donkey Kong*, *Donkey Kong Jr.*, and *Popeye*. Timing was very important, because the United States had just gone through a video game crash, largely due to the failure of Atari and other consoles (due in large part to poor management decisions that miscalculated demand). As a result of poor market conditions, Nintendo didn't release an American version of the Famicom until 1985. When Nintendo's first console did finally reach U.S. shores, Nintendo had 90 percent of the Japanese market. The NES quickly dominated the U.S. market as well. (In fact, Nintendo was so worried about U.S. sales that it marketed the Famicom as an educational computer system, complete with a talking robot.)

During the lifetime of the NES, hundreds upon hundreds of games were produced for this console. This system was released in Japan as the Family Computer, or Famicom, and featured the 8-bit 6502 microprocessor, with a clock speed of only 1,790 kilohertz (1.79

MHz!). Many popular American computers used this chip: Apple, Atari, and Commodore, among others. This is the console that established Nintendo as the firm leader in the console business, and the company poured its resources into this division, dropping out of the arcade business altogether.

The year 1989 saw the introduction of the first Game Boy, which would become the beginning of Nintendo's domination of the handheld market. The Game Boy was designed by Gunpei Yokoi, who had designed the Game ▤ Watch handheld games for Nintendo earlier in the 1980s.

Gunpei Yokoi was posthumously honored with a lifetime achievement award at the Game Developer's Choice Awards, presented by the International Game Developer's Association in San Jose on March 6, 2003, for his work on the Game Boy and for co-creating many blockbuster franchises such as Donkey Kong, Mario, and Metroid. With more than 140 million units sold worldwide, his Game Boy was the most successful video game system ever made. The award was accepted by the Yokoi family on his behalf. Rest in peace, Yokoi-san.

Console manufacturers measure the system, data, and storage capacity in bits, rather than bytes. To convert bits to the more familiar bytes format, simply divide a number by 8. For example, the Game Boy is equipped with 64 kilobits (Kbits) of memory, which equates to 8 kilobytes (KB). I use these terms somewhat interchangeably throughout the book, denoted by Kbit or KB.

The 16-Bit Era

In 1991 the 16-bit Super NES was released, and with it the fantastic *Super Mario World*. This time, however, Nintendo faced a strong rival in Sega, with its equally powerful 16-bit Genesis console and an equally popular *Sonic the Hedgehog*. In 1991 Sega released the Game Gear handheld video game system. In every respect, the Game Gear blew the Game Boy out of the water: graphics, sound, color, backlight, and processing power. But part of the problem with the Game Gear was battery life, something that made Game Boy more appealing. (About the same time, Atari released a color handheld system as well, called the Lynx.) And in usual Nintendo fashion, Game Boy showed that technology doesn't matter, games do. A simple and unassuming game called Tetris, bundled with the early Game Boy, helped Nintendo's handheld to quickly dominate the market.

Success and Failure

Facing ▪ serious competitor, Nintendo was at a crossroads in the early 1990s. The 16-bit consoles had a long potential life span, as had the 8-bit consoles. But anything could happen in the video game market during the estimated five-year lifetime of the Super NES. Nintendo really had no plan for the next system yet. The next logical step would be a

38

migration to 32 bits. While some sort of 32-bit Nintendo console was expected from everyone, Nintendo made a very strange decision for its next system. Virtual Boy is an example of what happens when marketing people—instead of game designers and developers—make the decisions in a video game company. This ugly mistake was driven by the popularity of virtual reality in the early 1990s, and it was a complete failure (as right it should have been).

A Detailed Comparison

Table 1.1 summarizes the history of the major video game consoles over the past 20 years.

Table 1.1	Console History		
Year	Manufacturer	Console	Bus
1985	Nintendo	NES	8-bit
1989	Atari	Lynx	8-bit
1989	Nintendo	Game Boy	8-bit
1990	Sega	Genesis	16-bit
1990	NEC	TurboGrafix 16	16-bit
1991	Nintendo	SNES	16-bit
1991	Sega	Game Gear	8-bit
1993	Atari	Jaguar	64-bit
1993	Sega	32X	32-bit
1994	Sega	Saturn	32-bit
1994	Nintendo	Virtual Boy	32-bit
1995	Sony	PlayStation	32-bit
1996	Nintendo	Game Boy Pocket	8-bit
1996	Nintendo	Nintendo 64	64-bit
1998	Nintendo	Game Boy Color	16-bit
1999	Sega	Dreamcast	128-bit
2000	Sony	PlayStation 2	128-bit
2001	Nintendo	Game Boy Advance	32-bit
2001	Nintendo	GameCube	128-bit
2001	Microsoft	Xbox	128-bit

Now, unless one has a strong bias for one console or another, it is interesting to note that Nintendo is the only company to have released 8-bit, 16-bit, and 32-bit consoles (Game Boys) successfully in recent years. I also find it interesting that Nintendo was the only manufacturer to produce both a 64-bit and a 128-bit console. It is evident from this list that Nintendo dominates the video game industry, even to this day. Despite the sales, growth, and popularity of other consoles, it must be noted that Nintendo's business practices are efficient and effective.

What Happened to the Atari Jaguar?

The story of the Atari Jaguar is one of frustration. The Jaguar, back in 1993, was similar to a Nintendo 64 and a PlayStation (both of which arrived on the scene a few years later) and was manufactured by IBM (the same company that designed and built the GameCube's central processor). The Atari Jaguar design team must have had fantastic dreams for this console, and I share that dream. With a two to three year lead on the competition, why didn't the Jaguar totally cream the competition?

Most certainly, it was a lack of solid third-party development commitments that spelled the downfall of the Jaguar, along with perhaps poor marketing, because Atari had such a huge lead on the competition, the Jaguar should have been more successful. So what happened?. Despite solid titles like *Wolfenstein 3D* and *Doom*, with an endorsement from id Software, this is ■ real-world example that demonstrates the most important factor in console development: Technology doesn't sell, games do. There were some notable games for the Jaguar, such as *Alien* vs. *Predator*, *Golden Axe*, and numerous arcade ports, but the established customer base of the SNES and Genesis held the Jaguar back.

The Importance of Goals

Do you have what it takes to follow in the footsteps of master game developers, to rock the establishment and invent new genres that challenge our comfortable list of game categories and give marketing people a headache? While you are learning the ropes, write as many copycat games as you can manage. If you are up to the task of reproducing *Super Mario Bros.* or *The Legend of Zelda: A Link to the Past*, do it. You will only master the genres and break them up by remaking the original genre-establishing games and coming up with ideas of your own. By the time you are done with this book, you will have the knowledge and know-how to reproduce most of the Game Boy games out there.

Use Your Imagination

Use your fresh insight and your imagination. Don't be concerned with telling others about your ideas, building ridiculous Web sites about your nonexistent game, and releasing the grossly mislabeled "betas" after finishing the title screen of a game. If you know your game rocks, then ■ great number of gamers will feel the same way after playing your game. But finish your game . . . completely . . . before even mentioning it! Do that, and you will gain unequalled respect by your peers, and perhaps even a few game companies. You will be taken seriously. (Many, many of the greatest in the business got started writing games for fun, not for profit.) You will surprise everyone. There's nothing, and I mean *nothing*, as cool as a brand new game released without any warning! Especially if that game is a lot of fun.

Build a Franchise

The first game released by id Software was *Commander Keen*, which was a PC shareware game. Ironically, *Commander Keen* has been ported to the Game Boy Advance! If you haven't played the game, I highly recommend it, because this is a great old-school platformer, created by John Carmack, John Romero, and Adrian Carmack. *Wolfenstein 3D* and *Doom* have also been ported to GBA, and John Romero is now producing portable games exclusively under the MonkeyStone label; his latest GBA game, *Hyperspace Delivery Boy*, is a top-view adventure game with a sci-fi theme.

Do you see any correlation between these PC masters and the console masters? Compare the works of John Romero with the works of Shigeru Miyamoto. Not every game is a smash hit (and there are even some real stinkers). But the theme of "franchise" is evident in games created by these masters of the game.

Genre- and Character-Based Franchises

Can you think of a theme that is brand new, has never been done before, that you can call your own? Create your own genre and build upon your brand-new characters, just as Miyamoto did with *Donkey Kong*, *Donkey Kong Jr.*, and *Mario Bros.*, and as he perfected with *Super Mario Bros.* Do you see how it took four games featuring the seminal "Mario" before Miyamoto had the characters, backgrounds, and foes he really wanted? (Alternately, you might build a franchise genre, as id Software did, starting with *Wolfenstein 3D* and on through several games to *Doom III* and beyond.)

You will see that even the latest *Super Mario Sunshine* features baddies from those old games, developed decades ago. Build your own franchise—that is the key to success in the video game industry. Think about all the great classics, and you will see a lineage, a dynasty, of unique characters. Think about *Sonic the Hedgehog, Castlevania, Mario, Zelda, Metroid, Bomberman, Contra,* and *R-Type.* The history is sometimes as much fun as the games themselves. If *Super Mario Sunshine* had been named something like "Mechanic Sunshine" and featured a wrench-toting little guy going around fixing the environment, do you think it would have been the same game? No, most of the sales for that game came from the title alone, because Mario games have a history of being fantastic. If you like Coke, and something new like Vanilla Coke comes out, you are probably going to like it regardless, because the familiarity of name and similarity of good taste appeal to memories of the fun you have had in the past with that product. This principle doesn't apply to every situation, of course. Some game franchises are becoming a scourge of sequel overkill. It's not great quantity that gamers are seeking, but gameplay.

I believe that if *Super Mario 64* had failed in gameplay, fans of the series would have been skeptical of *Super Mario Sunshine,* due to the 3D factor. *Super Mario 64* was a difficult game to master, and the camera was difficult and unwieldy (at least at first), but the gameplay fascinated a generation of gamers and formally ushered in the beginning of full 3D games (not to be confused with first-person shooters).

Strike a Balance: Level Count vs. Difficulty

How many genres has Miyamoto invented altogether? Quite a few, I would guess. *Super Mario 64* was a gamble, but the familiar gameplay of the original NES and Super NES games was translated into the third dimension perfectly. The character did feel like Mario, and this timeless classic is now considered by many as the greatest game of the 20th century. How could a great series like this have come from a weird (but fun) arcade game called *Donkey Kong?* Diversity of characters (such as plumbers and dinosaurs), creative themes (such as gathering coins), and familiar creatures (such as turtles) used as baddies.

And don't forget about the sound effects and music, which are just as endearing to the Mario dynasty as the graphics. My theory is that Mario games involve a *lot* of levels and baddies to beat, rather than a few difficult ones, and that is what makes them so much fun. Who wants to repeat the same level over and over again, without reward? (That gameplay decision alone turns me off to many, many games . . .). Difficulty level in a game can be balanced with the number of levels in the game. Increasing one usually involves decreasing the other, like a teeter-totter.

Surprise the Critics

What will you invent that people will be playing for decades to come? Grasp the concepts in this book as much as the programming aspects, and start inventing. Do the games first, and the career will come (if that is what you are seeking). This really flies in the face of career game developers who really have no love for games. Those people should get out of the way for real talent, because they are responsible for the many mediocre games we must wade through to find the gems.

Do you recall the feeling of discovering a new game that you have never played before, and how it blows you away because it is so much fun? After you finish the game, you desperately look for sequels or similar games or downloadable updates (which are becoming more common for consoles now with online capabilities). That is the market for copycat games. But the point is to try to grasp what makes those games so much fun and then emulate those concepts in your own games. Don't copy them pixel by pixel!

What I mean is, essentially, to learn about what works and what fails and then emulate the themes that work, combined with the machinations of your own imagination. If you love mindless shooting games like *Smash TV*, then do your own with variations on the theme, and make it ■ totally different game. Do this while learning the tricks of the trade, until such time that you are able to take hold of a truly unique and fun idea of your own. Here is ■ breakdown:

1. **Inspiration.**
   Inspired by a great game, you dream of making a similar game.
2. **Emulation.**
   Your first attempts are copycat games, as you develop your skills.
3. **Imagination.**
   Finally, you are able to incorporate new ideas into your own creations.

I will cover more game design aspects in a limited fashion throughout the rest of the book as this subject relates to each chapter.

Summary

This chapter presented an overview of the video game market, with a short history lesson about Nintendo and the developments that led to the Game Boy Advance, including an overview of the major consoles that have been released in the past two decades. This

chapter also covered some aspects of game design that might help aspiring game developers to focus their talent in productive ways. Finally, this chapter set the pace for the rest of the book, establishing the types of subjects and themes that will be focused on in following chapters.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The answers may be found in Appendix D, "Answers To The Chapter Quizzes."

1. What is the name of the company that created the Game Boy Advance?
    A. Sega
    B. Nintendo
    C. Atari
    D. Sony

2. Which of the following classic NES games have been upgraded for the Game Boy Advance?
    A. *Virtua Fighter*
    B. *Sonic The Hedgehog*
    C. *Super Mario Bros.*
    D. *Golgo 13*

3. How many megabytes are there in 384 megabits?
    A. 48
    B. 16
    C. 32
    D. 64

4. Who designed the game *Donkey Kong*?
    A. Hiroshi Yamauchi
    B. Gunpei Yokoi
    C. Yu Suzuki
    D. Shigeru Miyamoto

5. Who, in 1889, founded the company that would eventually become known as Nintendo?
    A. Shigeru Miyamoto
    B. Fusajiro Yamauchi
    C. Gunpei Yokoi
    D. Hiroshi Yamauchi

6. What does the word "Nin-ten-do", derived from Kanji characters, stand for?
   A. "Leave luck to heaven"
   B. "The video game company"
   C. "We sell card games"
   D. "Have yen, play game"

7. What was the Nintendo Entertainment System called in Japan?
   A. NES
   B. Mario Machine
   C. Family Computer
   D. Super Famicom

8. What was Nintendo's first arcade video game called?
   A. Donkey Kong
   B. Space Invaders
   C. Mario Bros.
   D. Computer Othello

9. Who invented the Game Boy?
   A. Gunpei Yokoi
   B. Shigeru Miyamoto
   C. Yu Suzuki
   D. John Romero

10. What is the system bus bit depth of the Game Boy Advance?
    A. 16-bit
    B. 32-bit
    C. 64-bit
    D. 128-bit

Chapter 2

Game Boy Architecture In A Nutshell

A solid understanding of the hardware specifications and capabilities of the Game Boy Advance is necessary in order to write the most efficient code for this platform. Programming any console is a rewarding experience because there are no layers between you and the hardware itself. When you modify a bit here and a bit there, things happen! There is no operating system, no game library, and in the case of the Game Boy Advance, not even the luxury of a graphics processing unit (GPU). Despite what might seem like limited hardware specifications, one must remember that when working directly with the hardware, without any layers, things move along very quickly. That is truly what makes Game Boy Advance programming so rewarding. A programmer's most natural habitat is as close to the hardware as possible, and consoles uniquely provide that environment.

This chapter presents an overview of the Game Boy Advance's hardware specifications, explaining how it works, or what makes it tick, so to speak. The display screen, memory architecture, and main processors are covered in detail. This chapter also examines previous Game Boy models, comparing and contrasting them with the Game Boy Advance. As a hardware reference and guide, feel free to refer back to this chapter at any time when you need some specifics on the hardware. The pace is somewhat fast, and I don't explain every single detail at this time, because many of these concepts are covered in later chapters.

Here is a summary of the subjects covered in this chapter:

- Game Boy handheld systems
- Direct hardware access
- Memory architecture
- The central processor
- Graphics and sound capabilities

Game Boy Handheld Systems

The Game Boy Advance has a long and fruitful history that goes clear back to 1989 when the original Game Boy came out. The Game Boy Advance is sort of the great-grandchild of that first Game Boy, because it is the fourth Game Boy model. New as it may be, however, the Game Boy Advance has now been supplanted by the Game Boy Advance SP. Granted, the internal hardware is architecturally the same, but this new SP model has some considerable new options, not the least of which is a backlit screen. Let's peruse all of the Game Boys that have made their way into our hearts over the years. Table 2.1 shows an overview of the Game Boy models and their specifications.

Table 2.1 Game Boy Specifications

Model	CPU	Memory	Display	Colors
Game Boy	8-bit Z80 4.17 MHz	64 Kbits	160 x 144	4
Game Boy Pocket	8-bit Z80 4.17 MHz	64 Kbits	160 x 144	4
Game Boy Color	8-bit Z80 8.0 MHz	384 Kbits	160 x 144	56
Game Boy Advance	32-bit ARM7 16.7 MHz	3,072 Kbits	240 x 160	32,768
Game Boy Advance SP	32-bit ARM7 16.7 MHz	3,072 Kbits	240 x 160	32,768

Game Boy, 1989

The original Nintendo Game Boy (shown in Figure 2.1) was released in 1989, only four years after the NES came out in the United States. Operating on four AA batteries, the Game Boy was not a revolutionary console by any means (an attribute shared with the NES), but it had a relatively long battery life. This first Game Boy was equipped with a Zilog Z80 microprocessor—the same one used on many electronic devices in the 1980s. In fact, all of the Game Boy models up to and including Game Boy Color featured a Z80 CPU, although later models were faster. The Game Boy's CPU runs at 4.17 MHz, which is comparable to the first IBM PC at 4.77 MHz. Not bad for a tiny little handheld!

The first Game Boy came with 64 Kbits of memory, which is a very limited amount! However, due to the small display screen, with a resolution of 160 x 144 and only four-color grayscale, very little memory was required for graphics. Four colors is really insignificant, memory-wise; that's what you might call 2-bit color. Since two binary digits can store the numbers 0, 1, 2, or 3, there are your four colors! Obviously, color 0 was black. That didn't

leave much for artists, so to state that the Game Boy had primitive graphics is exactly right on target.

Figure 2.1
Nintendo Game Boy, 1989.

To put this in perspective, the Atari 2600 had better graphics than the Game Boy, and it was a decade older. However, a very low memory footprint for 2-bit graphics does allow for some interesting games, even if the color support is limited, and this kept manufacturing costs down for Nintendo. A typical 8 x 8 sprite would use up only 128 bits of memory (that's just 16 bytes), and the 160 x 144 video buffer would have required only 5,760 bytes of memory (although there was no "video buffer", per se, on the Game Boy). As ∎ side note, the Game Boy was capable of handling only 8 x 8 and 8 x 16 pixel sprites, and only a maximum of 40 at a time.

Game Boy Pocket, 1996

The Game Boy Pocket (shown in Figure 2.2) was a slimmed-down version of the Game Boy. Although the hardware specifications were essentially the same, the Game Boy Pocket required less voltage to operate (3 volts instead of 6) and thus could be powered by only two AAA batteries, which are much smaller than AA and suited the small size of the Game Boy Pocket.

Figure 2.2
Nintendo Game Boy Pocket, 1994.

Game Boy Color, 1998

The Game Boy Color (shown in Figure 2.3) was significantly more capable than the previous two models and greatly aided game developers with a faster CPU and more memory, while still retaining compatibility with the older game cartridges. The Game Boy Color only requires two AA batteries and was therefore much lighter than the original Game Boy.

Figure 2.3
Nintendo Game Boy Color, 1998.

The Game Boy Color not only was capable of displaying 56 colors on the screen at once but also enhanced existing Game Boy games to 32 colors, greatly improving their appearance and playability. Grayscale games came to life on the Game Boy Color with multiple shades of color. Obviously, backward compatibility was an important factor for Nintendo, primarily for marketing reasons. Claiming that a just-released console (such as the Game Boy Color)

has a game library of several hundred titles is an impressive feat! Of course, a large percentage of those games are very low-quality Game Boy titles, due to the limited capabilities of the GB. Many excellent titles were released for Game Boy Color, including the phenomenally successful *Super Mario Bros. Deluxe* and *The Legend of Zelda: Link's Awakening DX.*

Game Boy Advance, 2001

The Game Boy Advance (shown in Figure 2.4) is significantly more powerful than any previous Game Boy model, with nearly twice the screen resolution, 10 times more memory than the Game Boy Color, and a blazing-fast RISC CPU (more than twice as fast as Game Boy Color). In addition, the Game Boy Advance incorporates the original Z80 CPU from the Game Boy Color, providing for complete backward compatibility with all previous Game Boy cartridges. I would surmise that Nintendo was primarily concerned with supporting Game Boy Color titles rather than the older grayscale Game Boy games, although all previous cartridges *will* work!

Figure 2.4

Nintendo Game Boy Advance, 2001.

Basically, what happens is that the Game Boy Advance detects the type of cartridge that has been inserted and boots up on either the ARM7 or the Z80 CPU, based on the cartridge. This ingenious architectural design allows the Game Boy Advance to run all previous games, all the way back to 1989—a significant achievement of electronics engineering. If you think about it, how many other consoles today are capable of running games from 1989—original games, in their original cartridges? None! Only the Game Boy Advance is capable of this feat (and the Game Boy Color before it).

Before getting on with the next model, the Game Boy Advance SP, I want to show you an awesome accessory for your GBA. If you already own a GBA and are considering purchasing a Game Boy Advance SP model only for the backlight, you have another option. Figure 2.5

shows a Game Boy Advance in normal indoor lighting with an Afterburner installed. This is an inexpensive internal flat-surface LED that is positioned in front of the LCD, providing a remarkable improvement in screen visibility regardless of the lighting conditions.

Figure 2.5
This Game Boy Advance is equipped with an Afterburner to brighten the screen.

Some soldering is required, but the work is relatively easy to do (that is, if you have any experience with a soldering iron—if not, you should ask a friend who is experienced with one to help you). For development work, the Afterburner is an essential add-on. It is very inexpensive (under $30) and may be ordered from online Game Boy Advance retailers, such as Lik-Sang (http://www.lik-sang.com). I suggest this alternative because I prefer the original Game Boy Advance design.

Game Boy Advance SP, 2003

The Game Boy Advance SP (shown in Figure 2.6) is a variation of the Game Boy Advance with an internally lighted screen, long-lasting rechargeable battery (built in), and folding clamshell design. In all other respects, the SP has the same internal components as the Game Boy Advance, and the changes are cosmetic. While the rechargeable battery is internal (and therefore not replaceable without disassembly), it does promise longer life than the AA batteries used in a Game Boy Advance (and there are rechargeable battery packs for Game Boy Advance as well). One interesting aspect of the SP is that, when the screen is opened, it resembles the original Game Boy!

Figure 2.6
Nintendo Game Boy Advance SP, 2003.

Direct Hardware Access

The Game Boy Advance is a video game console, and yet it is similar to a PC in many ways. Both have one or more processors, random-access memory (RAM), a display screen (or monitor) with many different video modes, a digital signal processor (DSP) for sound, and some form of intuitive user input. Both the Game Boy Advance and a PC have a motherboard with a power supply and a basic input/output system (BIOS) chip that causes the hardware to boot up.

A PC provides access to the hardware primarily through the operating system, while a console primarily operates by storing all system functions inside the executable program (the game). The Game Boy Advance has no operating system. Game Boy Advance games have complete control over the hardware, at the lowest level. This gives the programmer a great deal of control over the console, but also great responsibility. Many software engineers specialize in applications, operating systems, network communications, or device drivers. As a Game Boy Advance programmer, you will touch on all of these areas and more, each time you write a game, because no one has paved the way, so to speak. Each new program you write for the Game Boy Advance must incorporate all the code necessary to display things on the screen, play sound effects and music, and detect button presses. Most of these features are programmed using direct memory address functionality.

On the PC, there are interrupts provided by the operating system that you can use to make things happen. On a GBA, however, you make things happen by reading or writing a number

to a specific portion of memory (called a *hardware register*). The Game Boy Advance changes instantly when you make such a change, because those memory addresses are directly tied to the hardware. If you turn on a bit somewhere in video memory, the screen will change to another video mode or perhaps even display a pixel.

As is usually the case when charting new territory, it is useful to draw a map along the way. Fortunately, someone has already provided all the memory addresses for us, so we don't need to fire numbers into random memory locations to see what happens—in fact, that would likely have an adverse effect on the Game Boy; who knows, it could even be damaged by it. Let's look at the general features of the Game Boy Advance to get an idea of the terrain ahead.

Memory Architecture

You may be aware that your PC has three distinct types of memory. You have your main RAM, which holds all the programs and data you're actively working with (ignore virtual memory for now since this looks to your programs like lots of main RAM). There is the hard disk, which stores information for long periods. And there is display memory on your video card.

The Game Boy Advance has similar kinds of memory. Like the PC, each address refers to a single 8-bit byte (which means that it is byte addressable). Also like the PC, the ARM ("Advanced RISC Machine") processor in the Game Boy Advance can access 8, 16, or 32 bits at a time. Things are a little more complicated, though, and you need to understand a little more about how the different parts are used. Table 2.2 lists the types of memory accesses the memory allows and the wait states each access incurs.

The CPU reads memory by sending an address to the memory at the start of a cycle and then reading the data at the end of that cycle. Ideally, the memory is fast enough to provide the data that quickly. Slower memory tells the CPU to wait for one or more additional cycles before reading the data. These extra cycles are called wait states. The fastest memories, such as IWRAM and VRAM, have no wait states (abbreviated 0WS) and therefore return data in a single cycle. EWRAM is 2WS memory, returning data in three cycles—one normal cycle plus two wait states.

Table 2.2 Game Boy Memory Access

Memory Type	Access Widths	Comments
IWRAM	8, 16, 32	32 KB "internal" working RAM. Typically used for fast scratchpad RAM and for time-critical code.
EWRAM	8, 16	256 KB "external" working RAM. Typically used for main data storage and multiboot code.
VRAM	8, 16	96 KB video RAM. Stores all graphics data. Can only write 16 bits at a time.
ROM	8, 16	ROMs can be read in either slow (4/2) or fast (3/1) mode. See chapter text for more details.
Game Save RAM	8, 16	The game save RAM is part of the cartridge. See chapter text for more details.

Internal Working RAM

Internal working RAM (IWRAM) is the only memory directly accessible on the 32-bit internal data bus of the CPU core, because it is actually built into the CPU itself. This is why it's called *internal WRAM* as opposed to external WRAM (covered next). IWRAM is the fastest memory in the Game Boy Advance and is also the only memory that can be accessed 32 bits at a time. The speed and width of this memory makes it ideal for running ARM code at full speed. Unfortunately, there are only 256 Kbits of this memory.

As far as the ARM processor is concerned, a word is 32 bits, while a halfword is 16 bits, and a byte is 8 bits.

External Working RAM

One might think something named external working RAM (EWRAM) would on a cartridge or something. However, it is built into the GBA. It's called *external* because it sits outside the CPU's core on the 16-bit data bus. We've got 2,048 Kbits of this to play with. This RAM, with each access taking three cycles, is slower than IWRAM.

EWRAM is where you will store large data items. You may cache graphics here before transferring them to VRAM for display. You can also place programs here using the multiboot

protocol. Programs will run a little faster here than in fast ROM and quite a bit faster than in slow ROM.

Cartridge Memory

The game cartridge contains the game's program and data stored in ROM. This is much like a CD-ROM for your PC in that it cannot be changed and is typically larger than the RAM you have available. Some cartridges also contain memory for saving games. Special cartridges, such as those made by Visoly, have memory called flash cartridges. These devices look to the Game Boy Advance like normal game cartridges and yet can be programmed with your own data much like a hard disk in your PC, using a flash linker device (which is covered in the next chapter).

The amount of memory that can be addressed by a particular number of address bits is always a power of two. You can address 2^{16} (or 65,536) locations by using 16 address bits. The closest power of two to 1,000 is 2^{10} (or 1,024). This has become the usual meaning for 1 KB in measuring memory. Therefore, 32 KB = 32 x 1,024 = 32,768 bytes. Similarly, 1 MB = 1,024 K = 1,048,576.

Game ROM

Like EWRAM, the ROM is accessible via the 16-bit data bus. There are two speeds at which ROMs can operate and two modes in which they can be accessed. Speeds for the ROMs are given as a pair of wait-state values, such as 3/1. The overriding factor is whether each access to the ROM can be classified as sequential or nonsequential.

A nonsequential access occurs whenever a new area of the ROM is read. Sending the memory address to the ROM takes extra time, and these accesses take the number of wait states indicated by the first number. In this example, the nonsequential access will take four cycles (three wait states plus the normal cycle).

A sequential access occurs when the very next access to the ROM is at the next address. In this case the ROM already has the next address available and only takes the smaller number of wait states. This use of sequential accesses means that a consecutive sequence of instructions that have no other data accesses can run with only 1 wait state for faster ROMs. Even slower ROMs (running with 4/2 wait states) can equal EWRAM speed for these short bursts, while fast ROMs can outpace EWRAM during such a run. What this means, basically, is that a game cartridge is capable of slow-mode and fast-mode access.

On average, however, even fast ROMs will fall behind EWRAM because long runs of sequential accesses are not the norm. There is a prefetch buffer in the memory controller

that allows some sequential accesses have no wait states. Unfortunately, this speed comes at a cost in power usage.

Bytes, Words, and Halfwords

The ARM processor uses some terminology for memory sizes differently than in the PC world. To understand why, let's look at the history of some of the terms.

Most of us are used to a *byte* being 8 bits, and indeed the ARM doesn't change this. Looking back in history, however, we find that a byte actually refers to the size of a native character for the computer's output device.

Just as the byte has had different sizes, a computer *word* is commonly defined as the normal amount of memory the computer processes at one time. In the early 1980s, IBM designed the IBM PC around the Intel 8086 and Apple built the Macintosh with the Motorola 68000. Both of these machines processed 16 bits at a time and used 8-bit ASCII character sets. These two machines have cemented the terms *byte*, *word*, and *double-word* as meaning 8, 16, and 32 bits, respectively, for most of personal computerdom.

For more than a decade, Intel and Motorola microprocessors have processed 32 bits at a time in their normal operations, and new 64-bit processors are now available for PCs (such as the AMD *Opteron* and *Athlon 64*). The word size of 32-bit computers is therefore 32 bits, while ■ word on ■ 64-bit processor is 64 bits. The terminology surrounding software for them, however, has maintained the older terms because the operating system APIs and data structures have evolved using the terms rooted in their 16-bit ancestors. But the important factor to remember is that a word usually comprises the same number of bits that are handled by the processor natively, and the byte has remained to this day a fixed 8-bit value.

The ARM has no 16-bit predecessor and need not retain backward compatibility with any overriding architecture (as is the case with the x86), although the Game Boy Advance does include the Z80 for backward compatibility. Imagine a modern PC with an old 80486 chip included along with a newer processor! ARM terminology follows the normal definitions for byte and word: an ARM byte is 8 bits, while an ARM word is 32 bits, while a 16-bit number is called a *halfword*. Throughout this book, I avoid the confusion by simply calling memory addresses and variables by their bit depth.

Game Save Memory

Some cartridges (including the Visoly flash cartridges) have nonvolatile memory for storing saved games on the cartridge. There are currently three different kinds of memory used for this: battery-backed static RAM (SRAM), Flash ROM, and serial EEPROM. Each type has unique methods of access. Some of the Visoly cartridges support all three types of game save memory, while some support only SRAM and Flash ROM (because of the special writer needed for an EEPROM).

Graphics Memory

There are three sections of memory that deal exclusively with video memory and the display screen: video memory, palette memory, and object attribute memory (OAM, for handling sprites).

Video Memory

Video memory is where all graphics data must be stored for display on the screen. VRAM is zero wait-state memory like IWRAM but sits on the 16-bit data bus, so you can only move data half as fast as in IWRAM. How VRAM is used depends greatly upon the video mode and other features that your program selects. One property of this RAM that I will point out many times is that it can only be written 16 bits at a time (while the bus is capable of ▪ full 32 bits). Trying to write a byte will actually write 2 bytes of the same value. This seeming flaw can actually be useful in certain circumstances, such as the ability to quickly redraw the screen.

Care must be taken when writing to VRAM during the time when the screen is being drawn. Attempting to change memory that is being used to draw the screen can result in graphics glitches and image tearing (and event where the image is being drawn while the screen is also being refreshed, resulting in an uneven image). Furthermore, VRAM accesses during screen time can be delayed while the CPU waits for the video hardware to perform its accesses.

Palette Memory

Most of the Game Boy Advance's video modes use palettes to specify the colors being used. The Game Boy Advance has two separate 256-color palettes: one for background images and one for sprites. Each of these palettes is further divided into 16 palettes of 16 colors in some modes, allowing graphics data to be compacted even more. Color 0 of any palette is

defined to be transparent no matter what value is actually stored in the palette memory. Palettes are usually updated during the vertical blanking (VBlank) period, during which time the screen is not being drawn. There is also one video mode that doesn't use a palette but directly addresses colors in each pixel (mode 4).

Object Attribute Memory

Object attribute memory (OAM) is where you store the attributes, or descriptions, of what a sprite is to display, how, and where. The actual graphics data comes from VRAM, but the sprite's position, size, and other information come from the OAM. OAM is commonly updated during VBlank, and one often mirrors this data set in main memory for improved speed.

The Central Processor

The processor in the Game Boy Advance is an ARM7TDMI chip. This is a 32-bit RISC processor with a three-stage pipeline, a hardware multiplier, and lots of registers—it is quite versatile. I can recommend two reference books for more detail about the ARM, if you are looking to get into some serious low-level programming, or if you are just curious.

- *ARM Architecture Reference Manual*, edited by David Seal, Addison-Wesley, ISBN 0-201-73719-1. Also known as the *ARM ARM*, this is a detailed description of the many ARM processors in use today. This book is invaluable when working in assembly language.
- *ARM System-on-Chip Architecture* (2000), by Steve Furber, Addison-Wesley, ISBN 9-201-67519-6. This book gives more of the how and why about using the ARM. Whereas the *ARM ARM* is the definitive reference book, Furber gives a more readable text and fills in some of the details about why things were done the way they were.

Two Instruction Sets

The ARM is a RISC (reduced instruction set computer) processor design. As with most RISC processors the instruction set is very regular, meaning that there are few ways to encode an instruction. RISC design makes it very easy to build a fast and inexpensive CPU. It does not, however, lead to very compact code. In general, *code density*—the number of instructions per unit of memory—is lower in RISC designs than it is in CISC (complex instruction set

computer) designs. While the ability to understand assembly language is a valuable skill when writing console code, I don't use it very much in this book, aside from ▪ primer in chapter 11, "ARM7 Assembly Language Primer." I have only included this chapter to help you interface with assembly, not to teach you full-blown assembly language programming, because this book focuses on the C language. Knowing how to interface with assembly routines can be helpful, so that is the focus of that chapter.

The designers of the ARM decided that they could create a denser instruction set—which they dubbed the Thumb instruction set—by cutting out some features and making the instruction encoding a little less regular. They did this in a way that allows the two instruction sets to work together. The CPU essentially converts Thumb instructions into their equivalent ARM instructions on the fly. Each Thumb instruction is 16 bits, whereas the ARM instructions are 32 bits. There is a performance benefit to using 16-bit instructions in a computer with 16-bit memory, although I don't get into Thumb mode at all in this book, as it is an advanced topic. True, there may be cases for using Thumb instructions to speed up parts of a game, but there are also cases for using regular ARM instructions as well.

CPU Registers

The ARM has 16 registers accessible at any time. A register is a physical component of the processor, and may be thought of as part of the "thinking" component. One of these registers, called R15, is the program counter, which keeps track of the current instruction being executed. A couple of the other registers have defined uses for some instructions. All the registers hold 32 bits each.

There are also some additional registers that are only used under certain conditions. There are, for example, registers that replace R13 and R14 (usually the stack pointer and link register) when interrupts occur. There are exceptions to most of these, but knowing this list will help you when looking at compiled code or reading through assembly language examples.

The ARM Procedure Call Standard (APCS) defines a convention for compilers to use when calling functions. This is the way the C compiler calls functions (except under certain optimizing conditions). You don't have to follow this convention when you write your own assembly functions, but you can't call those functions from C if you don't. Table 2.3 details the register uses for APCS. When one procedure calls another there is an assumption that

some of the registers will not be changed by the called procedure. The called procedure must save and restore these values if it needs to change one of these registers.

Table 2.3 ARM Procedure Call Standard

Registers	Usage
R0–R3	These registers are used for passing parameters to functions. Any parameters that don't fit here get passed on the stack.
R4–R10	These registers are used primarily for register variables. Registers 9 and 10 are also used for stack manipulation when switching modules, but you'll seldom, if ever, have to worry about this.
R11	This is the frame pointer. This register is typically set and restored during the prologue or epilogue code since it is the pointer through which all local variables are accessed.
R12	This is the interlink pointer. For most Game Boy Advance code this is a scratch register.
R13	This is the stack pointer. This points to the last item pushed on the stack. The stack is a "full descending" stack meaning that the stack grows downward (toward lower addresses) in memory and the pointer always points to the next item to pop from the stack.
R14	This is the link register. This register holds the return address for the subroutine. This register is often pushed on the stack and then popped directly into the program counter for the return.
R15	This is the program counter. You only directly change it to execute a jump.

The Graphics System

Truly the most important aspect of a console is the graphics system and its capabilities. The Game Boy Advance has an intriguing selection of possible video modes from which to choose. There are three tile-based modes that resemble the previous Game Boy graphics systems. In addition, there are three new bitmap-based modes that provide more creative freedom with a game.

It is important to note some specific numbers that don't change and thus may be relied upon from one Game Boy Advance unit to the next. The refresh rate equates to 280,896 clock cycles per frame (the time it takes to display the entire video buffer), and this provides a refresh rate of 59.73 hertz (Hz). Therefore, the maximum frame rate on the

Game Boy Advance is ~60 FPS. Any game or demo that claims to achieve more than 60 FPS is simply subframing the display, which may have practical uses for special effects, but in general this is an upper limit on the frame rate of the screen. Most consoles aspire to such a frame rate, so don't take this number by any means to reflect poorly on the GBA. It is, in fact, a significant refresh rate.

Tile-Based Modes (0–2)

There are three tile-based modes, as mentioned previously. A "tile" is a small 8 x 8 section of the screen, and this is how the first Game Boy handled all backgrounds. Of course, you may still use sprites that move over the tiled background. I will summarize each of these video modes in the following subsections.

Mode 0

In this mode four text background layers can be shown. In this mode backgrounds 0-3 all count as "text" backgrounds and cannot be scaled or rotated. Check out the section on text backgrounds for details on this.

Mode 1

This mode is similar in most respects to mode 0, the main difference being that only three backgrounds are accessible—0, 1, and 2. Backgrounds 0 and 1 are text backgrounds, while background 2 is a rotation/scaling background.

Mode 2

Like modes 0 and 1, mode 2 uses tiled backgrounds. It uses backgrounds 2 and 3, both of which are rotate/scale backgrounds.

Bitmap-Based Modes (3–5)

There are three bitmap-based modes, also mentioned previously. These are the more familiar video modes that resemble those found on a PC, with a given resolution and a video buffer. I will summarize each of these video modes in the following subsections. You don't need to remember all of this information right now. It will become second nature to you in time, as you actually use these memory addresses and so forth in actual code.

Mode 3

This is ■ standard 16-bit bitmapped (nonpaletted) 240 x 160 mode. The map starts at 0 x 06000000 and is 76,800 bytes long. See the color format table above for the format of these bytes. This allows the full color range to be displayed at once. Unfortunately, the frame buffer in this mode is too large for page flipping (a method of reducing flicker on the screen—covered in Part Two) to be possible. One option to get around this would be to copy ■ frame buffer from work RAM into VRAM during the retrace.

Mode 4

This is an 8-bit bitmapped (paletted) mode at a resolution of 240 x 160. The bitmap starts at either 0x06000000 or 0x0600A000, depending on bit 4 of the REG_DISPCNT register. Swapping the map by flipping bit 4 and drawing in the one that isn't displayed allows for page-flipping techniques to be used. The palette is at 0x5000000 and contains 256 16-bit color entries.

Mode 5

This is another 16-bit bitmapped mode, but at a smaller resolution of 160 x 128. The display starts at the upper-left corner of the screen but can be shifted using the rotation and scaling registers for background 2. The advantage of using this mode is presumably that there are two frame buffers available, and this can be used to perform page-flipping effects that cannot be done in mode 3 due to the smaller memory requirements of mode 5. Bit 4 of the REG_DISPCNT register sets the start of the frame buffer to ▊ x 06000000 when it is zero, and 0x600A000 when it is one.

The Sound System

The sound system in the Game Boy Advance comprises four FM synthesis channels for generating sound effects and music, primarily for backward compatibility. The Game Boy Advance also features two new 16-bit stereo digital sound channels capable of outputting sampled sound effects and music tracks. There is no built-in sound mixer for synchronous sound playback, so programmers must write their own sound mixers or use a third-party library. A sound mixer allows multiple sounds to be played at the same time. Conceptually, it does this by "mixing" them together, which is where the name comes from. Without ■ sound mixer, it is only possible to play one sound at a time, which is called *asynchronous playback*. I will cover the sound system in more detail in Chapter 9, "The Sound System."

Summary

The Game Boy Advance is truly an advanced handheld video game system that is worthy of the accolades it has received in the development community and by gamers themselves. Many of the best games of all time are being ported to Game Boy Advance because it has the processing power to handle high-end games that are either 2D or 3D. By understanding at least the high-level view of the Game Boy Advance architecture, you are able to better judge the type of game that is or is not possible—and then challenge those possibilities!

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The answers may be found in Appendix D, "Answers To The Chapter Quizzes."

1. What is the name of the new Game Boy model released in 2003?
    A. Game Boy Advance SP
    B. Game Boy SP
    C. Super Game Boy
    D. Game Boy Advance Pro

2. What processor was used in the Game Boy Color?
    A. 6802
    B. 8086
    C. Z80
    D. 68000

3. How much memory does the Game Boy Advance have?
    A. 8 KB
    B. 64 KB
    C. 128 KB
    D. 384 KB

4. When was the first Game Boy released in the United States?
    A. 1879
    B. 1983
    C. 1989
    D. 1991

5. What is the full name of the main processor inside the Game Boy Advance?

    A. 80386 DX4-100

    B. ARM7TDMI

    C. 68002

    D. 6501

6. What is the designation for the memory that is built into the processor?

    A. EWRAM

    B. VRAM

    C. IWRAM

    D. CRAM

7. What is the maximum frame rate of the Game Boy Advance as limited by the hardware refresh?

    A. 120

    B. 30

    C. 80

    D. 60

8. What are video modes 0, 1, and 2 called?

    A. Tile-based modes

    B. Bitmap-based modes

    C. Sprite-based modes

    D. Background modes

9. What are video modes 3, 4, and 5 called?

    A. Tile-based modes

    B. Bitmap-based modes

    C. Sprite-based modes

    D. Background modes

10. How many sound channels, overall, does the Game Boy Advance have?

    A. 2

    B. 4

    C. 6

    D. 8

Chapter 3

Game Boy

Development

Tools

Console development was always a difficult process in the past, requiring custom equipment and a special version of the console designed to run programs through a link cable attached to ■ special cartridge. These hardware kits were supplemented with custom software, along with a compiler, an assembler, and a linker, designed to generate binaries, another term for executable game code, to run specifically on that one console.

The manufacturer of a console designs the development kit, hopefully without making it too difficult to use, in order to attract developers to write games for the console. This chapter explains what home-brew solutions are possible and what hardware and software you will need to write Game Boy Advance programs without an expensive software development kit (SDK) or hardware interface.

Here is a list of subjects you will find discussed in the following pages:

- Game Boy Advance development
- Installing the development tools
- Introduction to the HAM SDK
- Running Game Boy programs

Game Boy Advance Development

Let's come to the point: The Game Boy Advance is an amazing little machine. You are eager to get started writing code, right? Early on in the Game Boy Advance timeline, it wasn't even possible for a hobbyist to write a Game Boy Advance program. Let's digress for a minute to explore how the grassroots Game Boy Advance (GBA) development community produced the tools that made this book possible.

Enterprising programmers were writing GBA programs at least six months before Nintendo released the first GBA unit. How, do you suppose, could that possibly have been done? It's an amazing story, actually. Console and video game fans have been writing emulators for years now for everything from classic coin-op arcade games. The popular emulation program MAME, short for *Multiple Arcade Machine Emulator*, can emulate any old-school arcade machine from the pre-3D era. The same talented group of programmers (I'm generalizing here) who write MAME and many other emulators got started on several GBA emulators very early—some as long as two years before the GBA came out! (How's that for unkempt anticipation?) An emulator is generally developed with the help of several ROM images, such as from a GBA game demo leaked from a third-party developer. Emulator programmers will reverse-engineer these ROMs by examining the codes inside ▪ ROM.

Deconstructing The ARM7 CPU

Here is where things get interesting! Using the specifications of the ARM7 processor, a programmer can deduce what the binary codes inside a ROM are supposed to do. Each byte inside ▪ ROM is significant and represents a binary instruction or piece of data. Since the ROM includes all the graphics and sound effects for the game, some sections of data are reserved in the ROM for data of this type. There are other kinds of data stored in a ROM as well, such as game levels, character stats, and even cheat codes. All of this information is stored in the ROM as a single "image" that is written to a ROM chip stored inside each game cartridge. The emulator will open the ROM file, read the image into memory, and start processing it. The task of the emulator programmer, early in the project, is to decipher the codes stored in the ROM of any new hardware system being emulated, including the GBA. Once the programmer has figured out how the ROM is divided up between instructions and data, the next step is to decipher the bytecodes, each of which represents a single instruction from the ARM7 instruction set.

I won't get into any assembly language so soon, but you may jump ahead to Chapter 11, "ARM7 Assembly Language Primer," if you are curious about it at this point (although I would recommend waiting). Some instructions consist of just 1 byte, while others take up 2, 3, or 4 bytes each in the ROM. The GBA hardware boots up by executing a single instruction at a fixed point in the ROM, which then starts the game running. There is no operating system, no special interrupts, no built-in code libraries—nothing. In a console, all the code is built into the ROM and there is no operating system, although that statement is only relevant to smaller and/or older consoles. Newer consoles do have an operating system built in, because most modern consoles are capable of doing more than just playing ROMs. For example, optical media consoles feature some sort of menu system in order to play CD music or DVD movies. You get the point, right?

Emulation: The Key to Reverse Engineering

The early GBA hobbyists gradually gained experience with Nintendo's proprietary as-yet-unreleased handheld video game console, and some very good emulators were developed long before the actual hardware was released. The legitimate hobbyists do the work of deciphering the ROM the hard way, without accepting any tips from licensed third-party developers, because it is a matter of pride, and the challenge of cracking the code, so to speak, is the whole point. Emulator programmers are among the most talented low-level software engineers in the world; they provide a great service to video game fans. Without emulation, most old arcade games would be lost forever, because the original arcade cabinets are usually not preserved over time. They are recycled for newer games, or they simply break down as the years pass and are then discarded as irreparable. Emulators allow us to play the great classics on a PC monitor; they even support gamepad-style joysticks.

What is the final recipe for the elite emulator programmer? The flash linker. I'll explore this fascinating hardware interface later in this chapter. Suffice it to say, there's no better way to perfect an emulator than by using a real game ROM. Using a flash linker, an emulator programmer can download his or her actual retail game cartridges through a special cable to the PC. Once a real ROM is available, it is then possible to fix all the quirks in the emulator. After all, with the real hardware and game available as a comparison, one can simply tweak the emulator until it runs the virtual ROM just like the real GBA hardware runs the actual cartridge.

Unfortunate Side Effects

Unfortunately, some unscrupulous individuals (that's geekspeak for *losers*) abuse the intended purpose of the emulators and pirate the game ROMs. I understand doing that with video games that are no longer in production (such as MAME ROMs). But Game Boy ROMs are still being sold in retail channels. Don't pirate retail games! Emulators such as VisualBoyAdvance were designed for running *new* code for the GBA, not *existing* code. Recognize that distinction! Yes, there are many professional GBA developers who are using tools like Visual HAM, Hamlib, and VisualBoyAdvance for actual retail GBA games. These tools might be in the public domain, but they are excellent tools that rival the official Nintendo SDK (in some respects). There are many GBA cartridges in stores today with code that was compiled with the very same compiler that you will use while reading along in this book! How do the professionals (and the hobbyists) test their new programs? Emulators.

Know the distinction, and preach it when you can. If you love your GBA, then discourage game piracy when you have an opportunity. Why should you care? It's a tradeoff that console manufacturers weigh when designing ■ new video game machine. In the case of the Dreamcast and GBA, for instance, an easier programming model was deemed helpful in order to attract third-party developers (such as id Software, Electronic Arts, Activision, Capcom, Square, etc.). Some consoles have a very difficult SDK to master, in comparison, such as Sony's consoles and Sega's early consoles. Some consoles, such as the Sega Genesis, Sega Saturn, and Nintendo 64, were legendary in programming circles for their difficulty. Piracy is usually negligible on closed systems during the peak sales period (the first four years of the console's life), while more open systems are cracked earlier. In the case of the GBA, it was cracked before it was even released! That was certainly not a tragedy; on the contrary, it shows the fervent zeal that fans have for the Game Boy systems overall. You are reading this book because you love your GBA, right? Or perhaps you are a professional

I do not condone the act of downloading game ROMs, whether they are owned or not. I simply disagree with the whole concept, plain and simple. Flash cartridges are prohibitively expensive for making backups of game cartridges, therefore this activity is a slippery slope that cannot be justified. I urge you to maintain a level of professionalism in this matter and purchase any and all games that you enjoy playing on your GBA. Let's face it, if you are reading this book, then you must have an interest in professional GBA development. Please don't undermine the company that created this fascinating and enjoyable machine; cartridge sales are the sole basis of income in the handheld market.

developer. Either way, you must admit that this is a fun system to play as well as to develop for.

What About Public Domain Development Tools?

I have only hinted about the development tools up to this point, so you know what they are, without really knowing any of the specifics. Later in this chapter, I will show you how to install and begin to familiarize yourself with HAM, the tool used in this book. If you are a professional developer and already using an official Nintendo SDK or another distribution such as DevKit-Advance, I challenge you to give HAM a whirl. As I have explained, Visual HAM is able to handle code designed for DevKit-Advance (or any other GBA toolkit based on the GNU compilers), and the editor and environment are very nice, with an almost obsessive amount of support by the authors of Visual HAM and the HAM SDK.

Visual HAM And The HAM SDK

As I mentioned in Chapter 2, there are some compilers and assemblers available for the GBA, because it uses the common ARM7 processor, which is also used in hundreds of other consumer electronics devices. This was a good move on the part of Nintendo, as far as keeping costs down by using a powerful and mass-produced processor. Nintendo obviously learned a lesson from the Nintendo 64, which used a custom chipset developed by Silicon Graphics. The Nintendo GameCube, for instance, uses an IBM PowerPC processor with an ATI GPU (graphics processing unit) that are almost identical to the retail processors available in stores! Likewise, the GBA uses an ARM7 processor and features the secondary Z80 processor for backward compatibility. Nintendo opted for familiar and proven technology this time around with both of its latest consoles. The only problem—for Nintendo, that is—is that the widely available tools for the ARM7 processor have made it possible for enterprising programmers (which is geekspeak for *hackers*, in case you weren't paying attention) to quickly adapt the existing tools for the GBA.

This is where things get interesting, because that is precisely what they did. The end result of all the effort that went into the early GBA development toolkits is HAM, by Emanuel Schleussinger, and Visual HAM, by Peter Schraut. These next-generation GBA tools incorporate programs written by several people, providing (in addition to VisualBoyAdvance) a graphics file converter and a complete sound system! (For more information about GBA sound, refer to Chapter 9, "The Sound System.") The one thing I want to emphasize is that you can use Visual HAM and the HAM SDK to compile and run any

GBA program based on GCC, including games developed under DevKit-Advance (another GBA distribution).

What Is HAM All About?

HAM is ■ distribution package that includes all the tools (including the C/C++ compiler) needed to write, compile, link, and run GBA programs. The term *distribution* comes from the fact that HAM uses the open source GCC compiler (which has a GNU license), along with an ARM7 assembler that was released to the public domain by the chip manufacturer, ARM. The HAM distribution is included on the CD-ROM in complete format that you may install to your hard drive. This includes all of the following tools:

- HAM Game Boy Advance SDK
- HAM code library
- Visual HAM development environment
- PE Map Editor
- VisualBoyAdvance emulator

These are all the tools you need to write fully compliant GBA programs that will run on the actual handheld. These tools were not written by one person, of course, but were developed by hundreds of programmers who have dedicated their skills to the open source community, which has made HAM possible. Despite the large number of people involved in GCC and the various GBA tools, the HAM SDK and HAM library were developed by Emanuel Schleussinger. Likewise, the Visual HAM development environment and PE Map Editor were written by Peter Schraut. I am grateful for their consent to feature these excellent tools in this book.

Visual HAM and the HAM SDK are capable of compiling any GCC-compatible GBA source code. There is no stipulation that you *must* use the HAM library (Hamlib) in your programs. Most of the code in this book is stock GBA, not Hamlib.

HAM on the Web

The Web site for HAM (http://www.ngine.de) is shown in Figure 3.1, while the Web site for Visual HAM (http://www.console-dev.de/visualham) is shown in Figure 3.2. I encourage you to visit the sites to get the latest versions of the development tools used in this book. As the development community tends to be a very dynamic and persistent medium for change, there will likely be new versions of the tools every few months. The version used in this book is HAM 2.7.
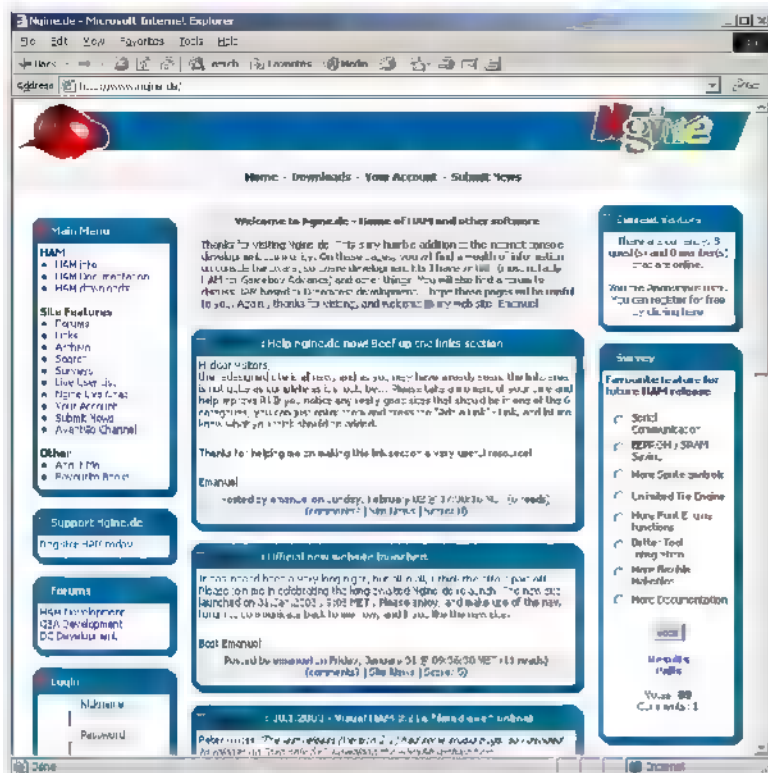
Figure 3.1

The Web site for HAM is located at http://www.ngine.de.
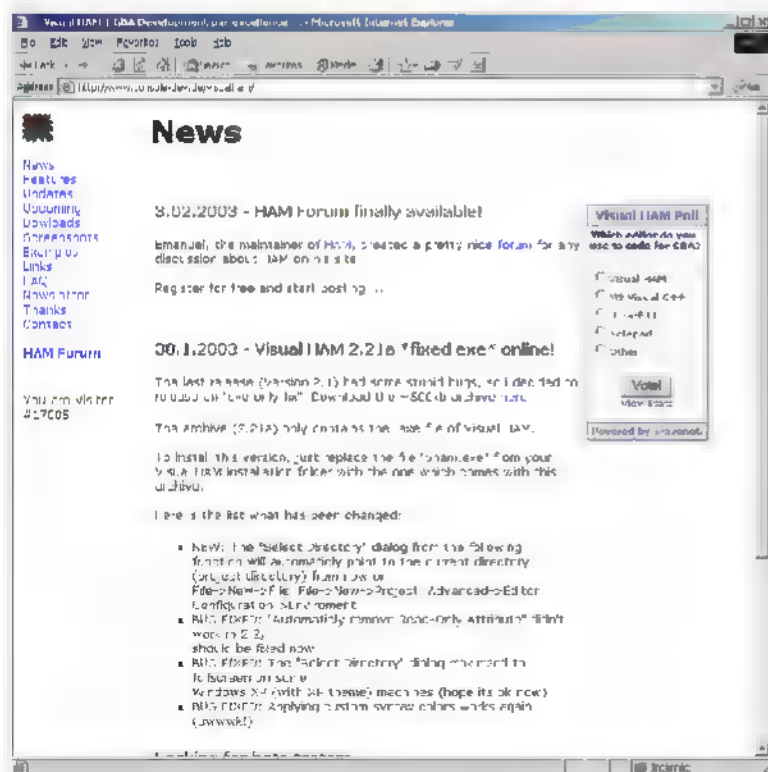
Figure 3.2

The Web site for Visual HAM is located at http://www.console-dev.de/visualham.

The Visual HAM Environment

Visual HAM was not developed by the same person who created HAM (Emanuel Schleussinger), although Peter Schraut works closely with him . The end result is ▪ seamless package (see Figure 3.3) that works so well that you will wonder why these tools are free! The exception is the HAM library (Hamlib). While you can write complete GBA programs without Hamlib, the library is a great help, because it encapsulates the whole GBA SDK within itself and handles much of the hard work for you, behind the scenes. I will show you in the next chapter how to write your first GBA program, with and without Hamlib. Just as an example and case in point, the source code shown in Figure 3.3 is just plain, simple GBA code, without any wrapper. (This program is actually a prototype of the graphics mode 4 pixel-plotting program that you will encounter again in chapter 5!).
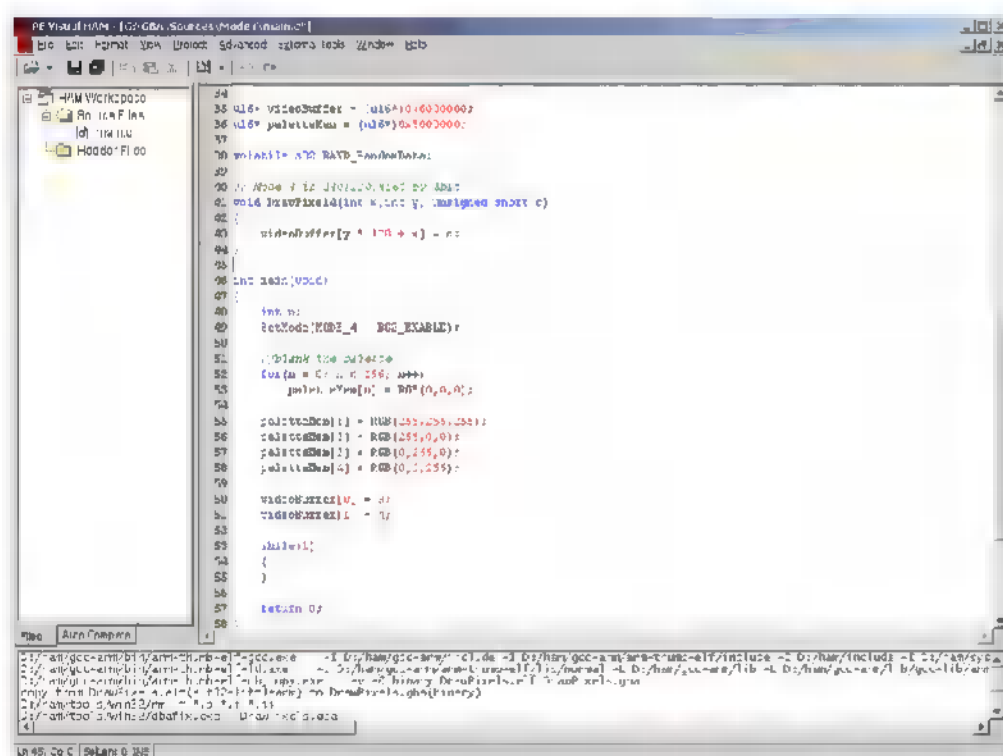
Figure 3.3
The Visual HAM integrated development environment.

It is important to know how to write core code without using any wrapper (which Hamlib provides), especially while learning. Although I use Hamlib to demonstrate how things work in the next few chapters, it is possible to use Visual HAM (the editor and integrated development environment) and HAM SDK to write non-HAM programs. I have downloaded several public domain GBA games written by fans, have loaded the source code into Visual HAM, and was able to compile and run the programs just fine, with no problems! The programs run in the VisualBoyAdvance emulator or on an actual GBA using a multiboot cable

(more on that in the next chapter!). In addition, Visual HAM comes with an excellent map editor that is useful for creating game levels, as shown in Figure 3.4.
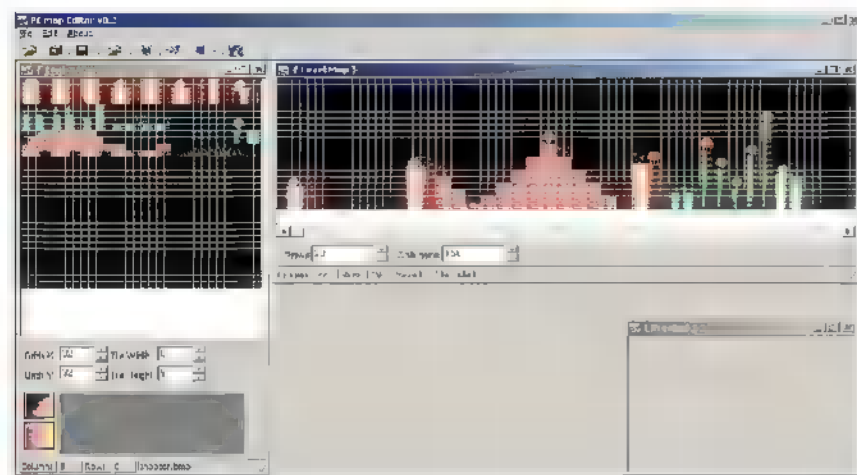
Figure 3.4
Visual HAM comes with a map editor for designing game levels.

Installing the Development Tools

The most remarkable thing about the HAM distribution is just how easy it is to install and use. HAM was designed to be a complete one-stop solution for writing and testing GBA code. As such, it comes equipped with everything you need to write, edit, debug, and run programs with several options for the output. HAM comes with an excellent GBA emulator called VisualBoyAdvance. Although Emanuel has packaged the emulator with HAM, you can still download VisualBoyAdvance from http://vboy.emuhq.com; be sure to check these links from time to time for updates to the tools. Although I have included the very latest on the CD-ROM, these tools are updated and improved over time by the GBA community.

Installing HAM

The downloadable zip file version of HAM has a file name something like ham-2.70-win32-full-distro.zip (note that the version number may change). Extracting the zip archive reveals two folders (ham-installer-files and ham-other), along with the setup file, setup.exe. The zip is available on the CD-ROM in the \HAM folder. Double-clicking on the setup.exe file starts the HAM installer, which is shown in Figure 3.5.
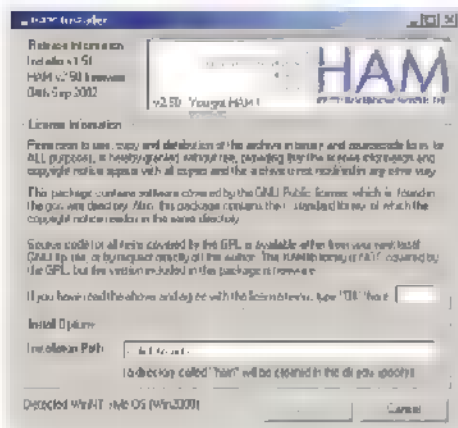
Figure 3.5

The HAM installer is a simple dialog-style window.

After typing "OK" in the box as requested, click in the Installation Path text box to bring up a folder selection dialog box. Select a hard drive where you would like HAM installed. Be sure to just select a root folder, not any subfolder, because HAM only works off the root.

HAM must be installed on the root of your hard drive! Although it will work fine on any hard drive (C, D, etc.), it must be on the root. For example, C:\HAM.

After selecting the pathname where HAM will be installed, the window should look like the one shown in Figure 3.6.
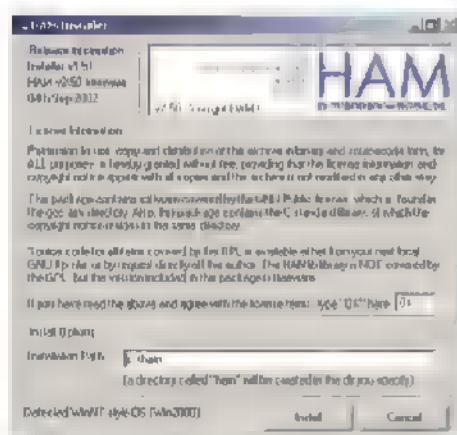
Figure 3.6

The installer dialog box now shows the target folder where HAM will be installed.

The HAM installer usually takes only a minute or two to install, after which the window shown in Figure 3.7 appears.
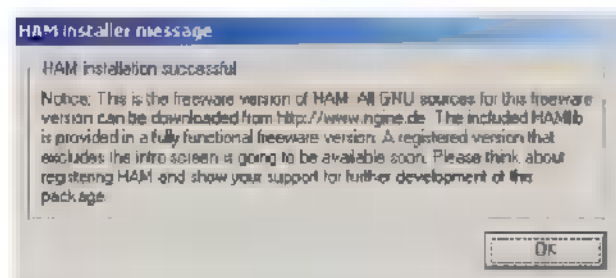
Figure 3.7

HAM has been successfully installed.

Installing Visual HAM

You may download Visual HAM from this Web site: http://visualham.console-dev.de. Or you may, of course, just install it from the CD-ROM that came with this book. Visual HAM doesn't require any special installer. It just comes as a simple Zip file, although it must be extracted with full folder structure intact. I have provided both the Zip file and the extracted Visual HAM folder on the CD-ROM. You may simply unzip the file to your hard disk drive or copy the folder from the CD-ROM, located at \Visual HAM. Note that the HAM SDK should be installed first, because Visual HAM looks for the HAM SDK when it first runs. Visual HAM is not very useful without the HAM SDK, because that contains the compiler, the assembler, the linker, and so on. Visual HAM is just the IDE, the attractive front end for these command-line tools.

So, assuming that the HAM SDK has been installed to C:\HAM, I would recommend copying the \Visual HAM folder from the CD-ROM to C:\HAM. That way, Visual HAM will be stored conveniently inside the HAM folder at C:\HAM\Visual HAM. Of course, there is no restriction to Visual HAM like there is with the SDK, so you may copy it to the root if you wish under C:\Visual HAM. What really matters is that you can easily find the VHAM.EXE file contained inside this folder, because I want to show you how to create a shortcut to it for your Windows desktop (a shortcut doesn't come with Visual HAM). After you have copied Visual HAM to your hard disk drive, minimize all applications and then right-click on your Windows desktop (anywhere) to bring up the right-click menu. Select New, Shortcut from the pop-up menu (as shown in Figure 3.8).
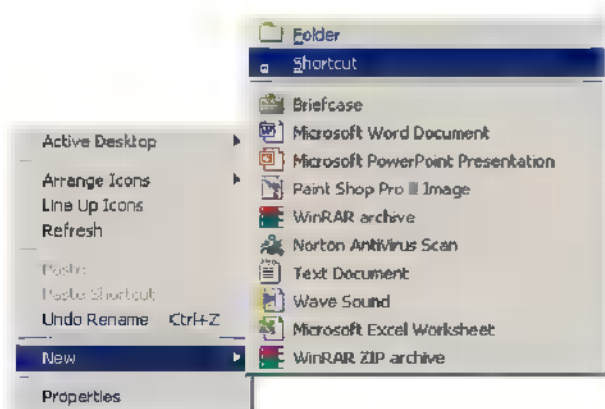
Figure 3.8

The right-click menu from the Windows desktop is used to create a shortcut.

The Create Shortcut dialog box will appear. Click on the Browse button and locate VHAM.EXE inside the Visual HAM folder that you copied to your hard disk drive from the CD-ROM (for example, see Figure 3.9).
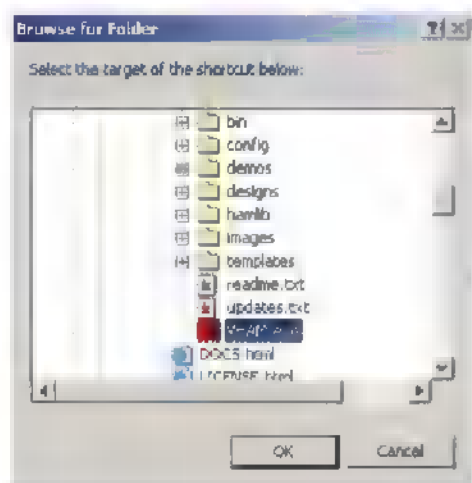
Figure 3.9
The shortcut browser is used to locate the VHAM.EXE file.

The selected executable file is pasted into the location field of the dialog box, as shown in Figure 3.10.
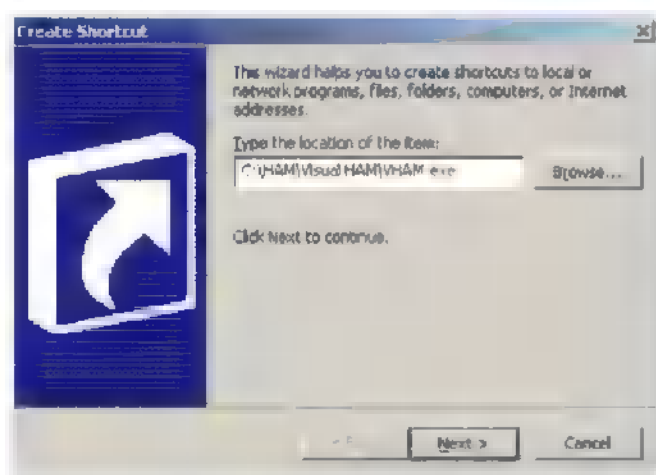
Figure 3.10
The executable file name is now shown in the location text box of the Create Shortcut dialog box.

Next, you can select a title for the new shortcut. I have entered the title "Visual HAM - Game Boy Advance IDE", as shown in Figure 3.11.
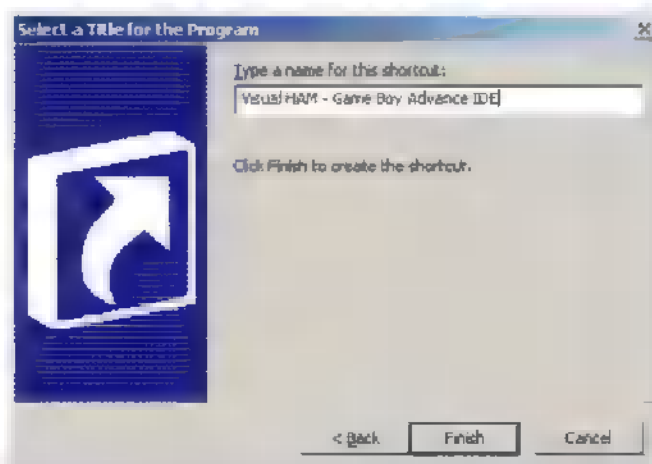
Figure 3.11
Typing in the title for the new shortcut to Visual HAM.

You now have a shortcut on your desktop for starting the Game Boy Advance development kit! Just double-click on the icon to open Visual HAM. The first time it runs, you will see a message stating that the program has configured itself and needs to be restarted. Just close Visual HAM, and then run it again, and you'll be all set and ready to go.

Configuring Visual HAM

The HAM SDK comes with a batch file called startham.bat that sets up the correct path for running the various HAM command-line utilities, as well as the compiler, assembler, and linker. You will need to run startham.bat anytime you use a command-line tool (such as the GFX2GBA.EXE program that you will be using frequently in coming chapters). However, for normal code editing and running of programs, you don't need to run the startham.bat file, because Visual HAM knows where all the tools are located. When you first install the HAM SDK, the installer creates a simple entry in the Windows Registry that specifies the version number and location of the HAM SDK. I will show you how to load a project into Visual HAM and run it in the next section of this chapter.

Once you have restarted Visual HAM, you are presented with a blank screen, as shown in Figure 3.12.
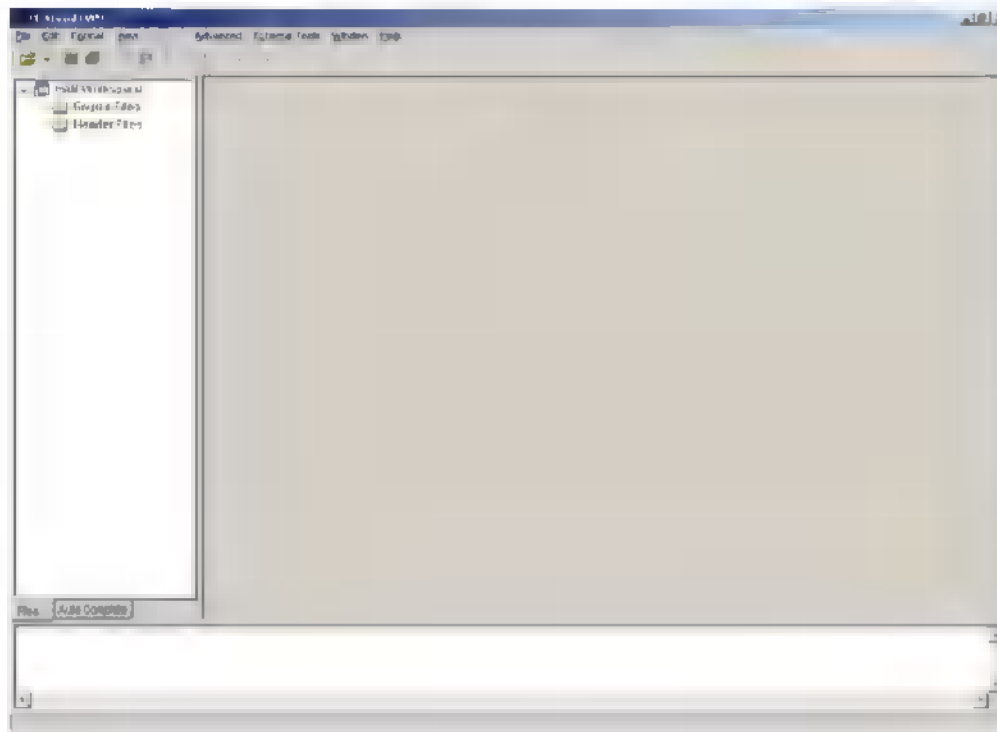
Figure 3.12
Visual HAM just shows a blank editor window and empty project when started for the first time.

Now, let's make sure everything is configured properly. Visual HAM has a dialog box that shows information about HAM. To bring up this screen, which is shown in Figure 3.13, open the Help menu and select Info about recent HAM installation. Note that in this case, it shows an install path of G:\ham\. I happened to have an unused hard disk drive partition that I decided to dedicate to Game Boy Advance programming, so that is why it shows the G drive here. Your installation will look different (most likely C:\HAM\). Take note also of the version, which is shown as 2.52 here. The version is arbitrary, and will change often as the development tools are improved (which is fairly consistently in the online community).

Version numbers change sometimes on a weekly basis in the development community, so just make sure you are using the latest versions of everything by browsing the HAM SDK and Visual HAM sites frequently, as well as the Web sites for support tools like VisualBoyAdvance and GFX2GBA. You will find the complete list of Web sites for these tools in Appendix B.
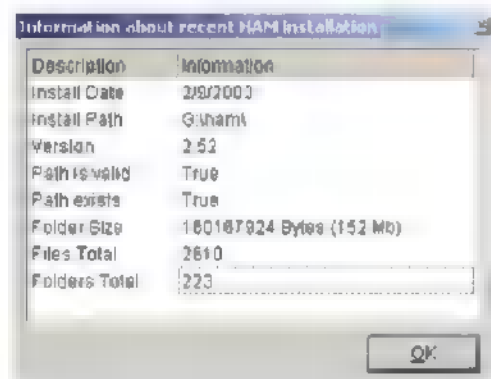
Figure 3.13

This dialog box shows information about the recent HAM installation, including version number.

Now, there is one more thing that I would like you to do, and then you will be finished with the installation. If you opened the Help menu a moment ago to display the HAM version information, you should have noticed a menu item called Associate 'vhw' Filetype. This is a very useful thing to do, so go ahead and click on that menu item. Nothing will be displayed, but Visual HAM actually just added an entry to your Windows Registry associating itself with all .vhw files (which stands for Visual HAM Workspace). You may now double-click on any workspace file directly from Windows Explorer to open Visual HAM. Trust me, you will be doing this a lot as time goes on, because it's far easier to double-click on a workspace file than to run Visual HAM, click on the Open icon, search for the correct folder, and then open the workspace. That's two clicks versus about seven.

Running Game Boy Programs

To demonstrate the capabilities of the HAM SDK and Visual HAM IDE, I've written a short program that displays a picture on the GBA screen. I will explain how to run this program in the VisualBoyAdvance emulator.

For starters, let me show you how to open and run this program, called ShowPicture, in Visual HAM. Please don't be concerned with the source code at this point. Although it's a short code listing, this program is for demonstration purposes only. I will explain the ShowPicture program in the next chapter (along with several more sample programs!).

First, I'll assume you have Visual HAM and the HAM SDK installed. Next, you will need to access some files on the CD-ROM that came with this book. Look for a folder called \Sources\Chapter3. Inside this folder you will find the project ShowPicture. If you haven't already, you will want to copy the project from the CD to your hard disk drive. (I recommend just copying the entire \Sources folder to your hard disk drive in order to gain access to all the projects in one fell swoop—be sure to turn off the read-only attribute on the folder and all subfolders and files.) This is necessary because Visual HAM (and the compiler) need to update files for the project as it is compiled and run. You can run the program directly off the CD-ROM by opening ShowPicture.gba, or by opening the ShowPicture.vhw (Visual HAM Workspace) file in Visual HAM.

If Visual HAM ever seems to stop compiling programs correctly, in that it no longer invokes the compiler, linker, and so forth, then most likely it is just not hooked up to the HAM SDK (for example, if you moved the HAM folder). There is an easy way to fix this. From within Visual HAM, open the Advanced menu and select Editor Configuration. Select the Environment tab, and then click on the Auto Detect button. Visual HAM should be able to locate the HAM SDK (using the Windows Registry) and fill in the root and path for HAM. This feature is likely to change in future versions.

Now that you have the ShowPicture project loaded into Visual HAM, let's try to compile the program. First, refer to Figure 3.14 to see what the project should look like on your screen. If Figure 3.14 looks the same as what's on your screen, then let's proceed. Otherwise, you may want to double-check the project you have loaded.
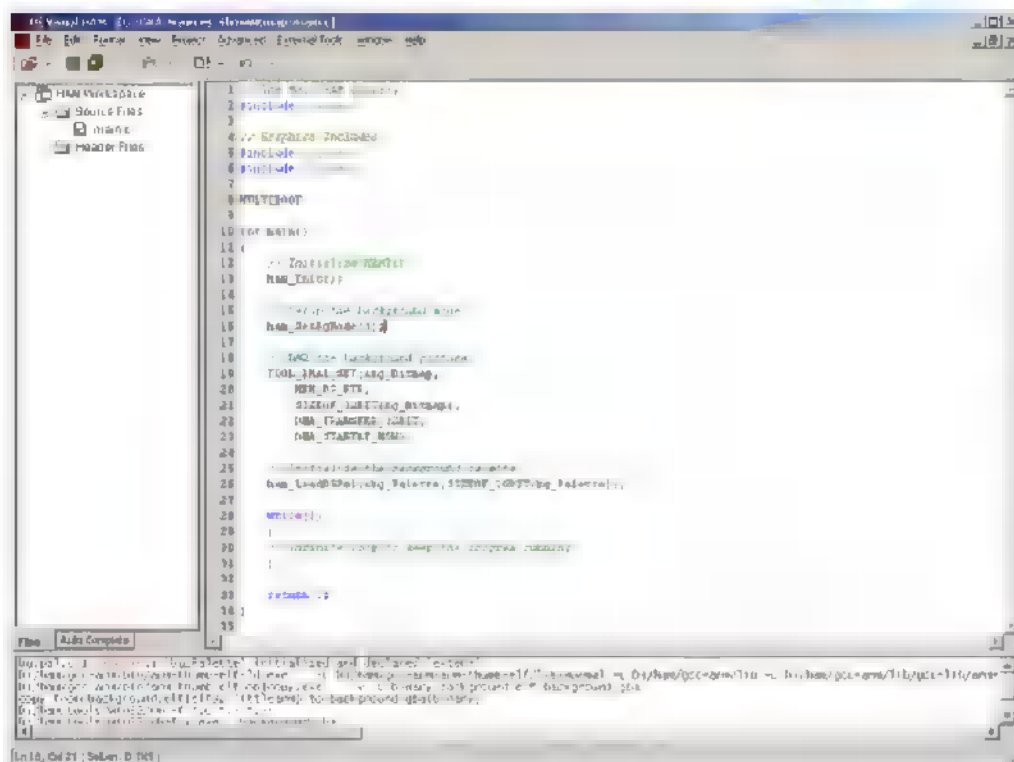
Figure 3.14

The ShowPicture project loaded into Visual HAM.

Now you can compile and run the program. To do this, open the Project menu, as shown in Figure 3.15. Select the Build + VBA option. This will compile the project and then run it in the VisualBoyAdvance emulator (which was included with the HAM SDK).

Figure 3.15

The Project menu in Visual HAM is used to compile and run programs.
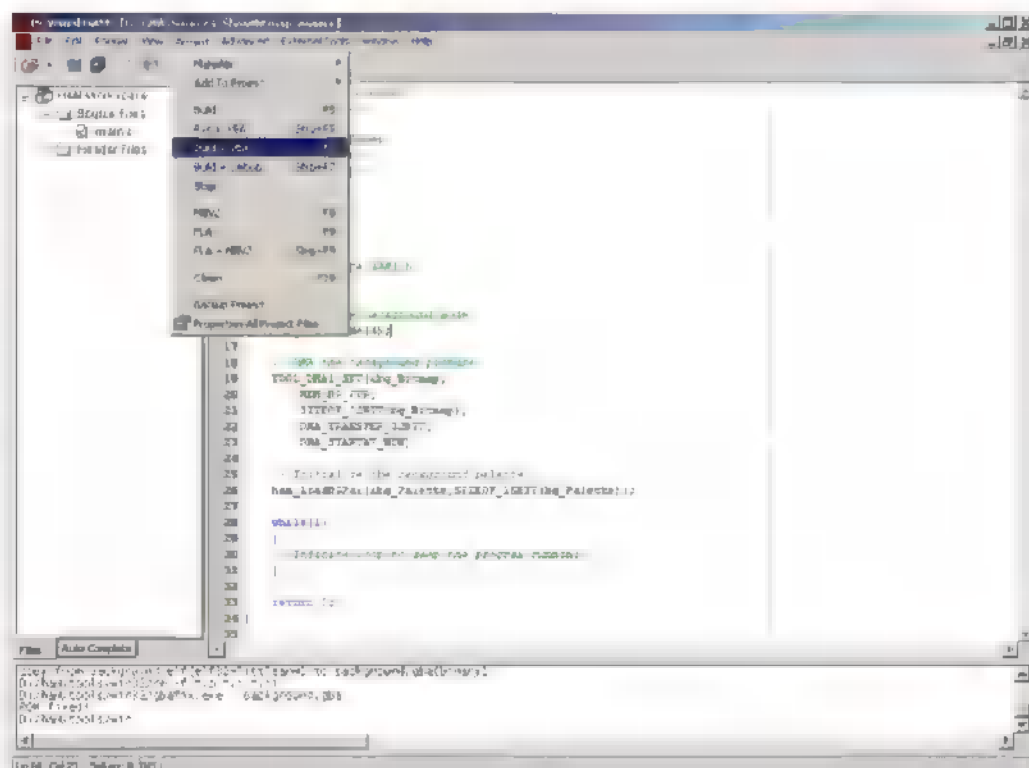
If all goes as planned, you should see the ShowPicture program start running in VisualBoyAdvance, as shown in Figure 3.16. How do you like that? You have just run your very first Game Boy Advance program, and it only required a few mouse clicks!
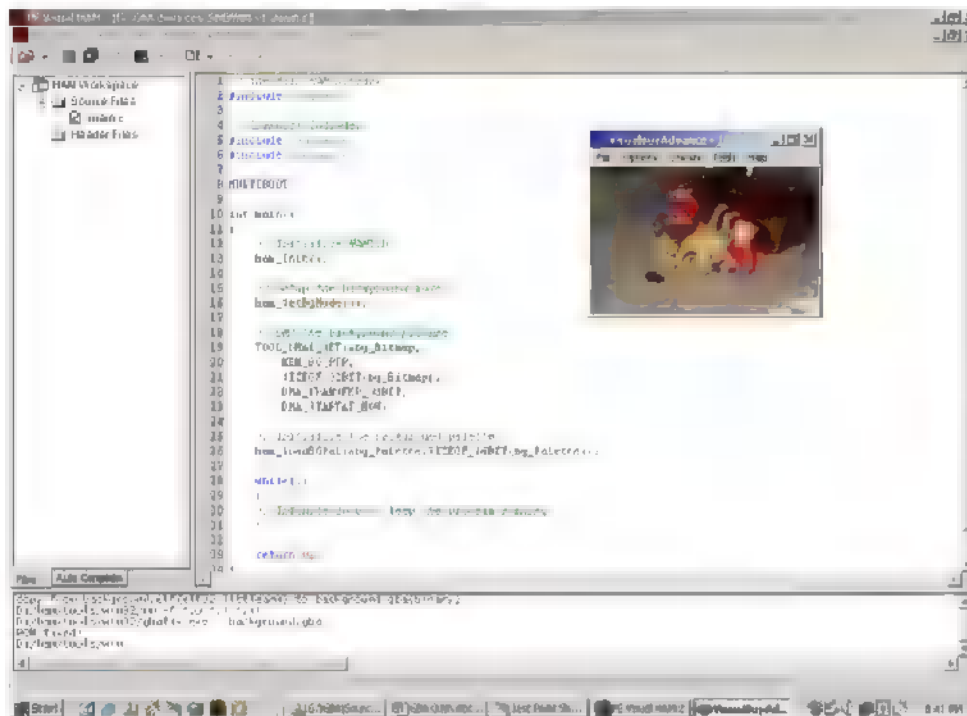
Figure 3.16
The ShowPicture program is running inside the VisualBoyAdvance emulator.

I don't know about you, but I was absolutely *thrilled* the first time I compiled and ran my first Game Boy Advance program! It's an incredible feeling, like taking that first step into previously unscathed terrain, where only a select few have trod. Most important, if you have made it this far, then that is positive proof that you have installed the GBA development tools correctly and may proceed through the rest of the book, running programs, writing code, learning secrets of GBA programming, and basically having fun!

Game Boy Emulation

VisualBoyAdvance is the emulator included with the HAM SDK, and the one I use in this book, although there are other GBA emulators available. The reason VisualBoyAdvance is preferred is because it includes some features for development, such as the ability to print out debug statements at the bottom of the VisualBoyAdvance window using the printf function (a standard C function).

As with all of these development tools, VisualBoyAdvance is constantly being updated with new features, more optimized code, and so on. The version that comes with HAM SDK is

probably not the latest and greatest, so I highly recommend that you visit the VisualBoyAdvance Web site (http://vboy.emuhq.com) frequently to check for updates. At the time of this writing, the latest version of VBA is 1.4, however, the version included with HAM SDK is version 1.0, and it is very limited. Later versions added some great new features, such as the ability to change the window size. I use this feature most often, because it is nice to see a larger version of the GBA screen; you will see sample screen shots using this feature in later chapters.

To upgrade the copy of VisualBoyAdvance that comes with HAM, you'll want to first download VisualBoyAdvance from the Web site shown above, and then extract the VisualBoyAdvance.exe file from the downloaded zip file. Now, the version included with HAM is just vba.exe, so you will want to rename VisualBoyAdvance.exe to vba.exe. After you have done that, you can copy the new vba.exe file over the old one. It is located in C:\ham\tools\win32.

Running Programs on Your GBA

In addition to emulation, there are also options for running your compiled programs on your very own GBA! There's nothing quite as fascinating as watching your own custom-written code running on an actual video game machine, even a small handheld like the GBA. This is where it's really at—where you can truly demonstrate your skills. While you may be just a hobbyist, there are many aspiring game programmers that I would like to speak to now. There is no better way to find a good job as a game programmer (and more specifically, a console programmer) than by showing a potential employer your demo programs running on a real GBA.

Now, one of the things I am recommending is that you produce a demo CD-ROM with VisualBoyAdvance and all of your precompiled GBA code, so you can send it to potential employers. This is not a book about finding a job, although I will give you pointers now and then, but I can tell you definitively that project leads and development studio managers and owners want to see demos before they see your qualifications on paper—assuming you have no experience, that is. Obviously, if you have experience in the game industry already, then you

Again, I am only talking about the hypothetical situation where you are an aspiring game programmer and you are honestly looking to get into the industry. If you are a hobbyist with no such aspirations, then please look over such advice columns and realize that there are a great many programmers in the world with *precisely* that kind of dream.

know what I'm talking about intrinsically. Nothing quite beats some actual demos that will tangentially demonstrate what you are capable of doing.

However, imagine getting an interview request after sending a demo CD-ROM and your resumé to ∎ potential employer. Why not take your own GBA to the interview with ∎ flash card containing all of your demo programs and games so you can show the interviewer your work firsthand? That is possible by using a flash linker, which is a device that plugs into the parallel or USB port on your PC and is capable of writing a ROM directly to ∎ flash cartridge (which houses flash memory, sort of like SmartMedia or CompactFlash cards). However, the flash cartridge looks like ∎ real GBA game cartridge and actually functions as one when plugged into a GBA. What a great way to impress an interviewer by showing him or her your demos running on a real GBA.

To impress a potential interviewer even *more*, consider installing an Afterburner backlight to your GBA (assuming you don't own a GBA SP model with a built-in backlight). The better your demos look on the screen, the better your chances for a second interview and/or a job offer!

What hardware options are available for running your programs on the actual device? There are two primary means of doing so: a multiboot cable and ∎ flash cartridge. The next two sections simply present an overview; you may look to the next chapter for ∎ complete tutorial on compiling and running programs with a multiboot cable or a flash linker.

Using A Multiboot Cable

Multiboot cables, such as the popular Multi Boot Version 2 (MBV2), take advantage of the multiboot capability of the GBA, where two or three players can connect to another GBA running ∎ game that supports multiple players.

What happens is that the other players are connected to the primary GBA using link cables, which are available at any video game store. GBA games such as *Mario Kart Super Circuit* with link cable support actually have small miniature versions of the game stored on the cartridge, and that mini-ROM is sent to the linked GBAs when they power up (note that no cartridge is needed for linked play). The multiboot cable takes advantage of the capability by fooling the GBA into thinking it is linking up to a multiplayer game, when in fact it is simply linking up to your computer. The MBV2 software detects the signal coming from the GBA, requesting the ROM, and sends the ROM to the GBA using the same protocol that the GBA uses when playing a multiplayer link game.

The GBA has only 256 KB of memory available for running multiboot games, which is a serious limitation for any large game project. However, many hobby projects (including the sample programs developed in this book) require less than 256 KB of memory, so they work fine as multiboot games. What happens, though, when you've written a really great game that takes more than 256 KB of memory? For instance, what if your game has a lot of cool graphics and sound effects and requires several megs of memory? A game that big won't work with the multiboot cable. That is a job for a flash linker.

Using a Flash Advance Linker

A flash linker is a device that reads and writes flash cartridges; there are such cartridges available that are compatible with the GBA game cartridge slot. This makes it possible to copy your compiled programs to a blank flash cartridge and then insert the cartridge and run your programs on the GBA—which is as close to the real thing as you can get. While ■ flash cartridge uses rewriteable memory, retail GBA games have PROM (programmable read-only memory) chips that are basically "write-once/read-many" in nature. Once ■ PROM has been burned, it is permanent. But flash memory is rewritable, and the technology is similar to EPROM (erasable programmable read-only memory), despite the apparent incongruity of terms. It actually does make sense when you consider that the target device can only read the memory chip, while a burner is capable of erasing and rewriting the contents of the EPROM chip.

A flash linker connects to the parallel or USB port on your PC and is capable of both reading and writing to or from the flash cartridge (which looks just like a regular GBA cartridge). There are some legal considerations when using a flash linker that are simply not relevant with the MBV2, because a flash linker can be used to copy a retail GBA game cartridge just as easily as it is able to read a rewritable flash cartridge. It is therefore possible to copy ■ retail GBA game ROM to a storage medium on a PC (such as the hard drive), which then makes it possible to make copies of the game. For this reason, the use of a flash linker is somewhat questionable and may not be legal in the country or state where you live.

I will assume you are a professional if you are reading this book and you have no desire to pirate games in this way. If you *are* interested in acquiring games by illegal means, I urge you to consider buying used games instead of copying them. For one thing, used GBA games are very affordable, while blank flash cartridges are very expensive (on the order or $150 and above). The implied consideration is also one that focuses on the concept of biting the hand that feeds you. If you are an aspiring game developer, show respect for the industry and discourage illegal software piracy by setting an example for your peers to emulate, and

demonstrate the benefits of purchasing legitimate games, with the obvious bonus of having ■ collectible game library.

Summary

This chapter was a critical step in the process of learning how to write GBA programs, providing an overview of the development tools, a tutorial on installing and configuring Visual HAM and the HAM SDK, and even a walk-through of loading and running an actual GBA sample program. As many GBA programmers have used various development tools (including the official Nintendo SDKs), this chapter helps to normalize expectations and establish the basis for the remaining chapters. Once it is understood what development tools are used to write GBA programs, the programmer may delve into later chapters knowing what to expect. For the beginner, this chapter was especially important and useful because arguably the most difficult aspect of getting started writing GBA programs is deciding on a development tool and then installing and configuring the compilers and other utilities. This chapter cleared up the issue and demonstrated the power and flexibility of the HAM distribution.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in Appendix B.

1. What is the name of the software development kit used in this book to compile Game Boy Advance programs?

    A. HAM

    B. Visual HAM

    C. Hamlib

    D. DevKit-Advance

2. What is the name of the integrated development environment used for editing source code and managing development projects?

    A. HAM

    B. Visual HAM

    C. Hamlib

    D. DevKit-Advance

3. What is the name of the comprehensive Game Boy Advance programming library featured in this chapter?
    A. HAM
    B. Visual HAM
    C. Hamlib
    D. DevKit-Advance

4. What does MAME stand for?
    A. Military Aftermarket Mobile Emitter
    B. Multiple Arcade Machine Emulator
    C. Multiple Arcade Maker Emulation
    D. Maintenance Associate Market Examiner

5. How much does a licensed copy of the HAM SDK and Visual HAM cost?
    A. $25
    B. $50
    C. $100
    D. Free

6. Who developed and released the HAM SDK into the public domain?
    A. Peter Schraut
    B. Emanuel Schleussinger
    C. Harvey Minefield
    D. Shigeru Miyamoto

7. Who developed and released Visual HAM into the public domain?
    A. Rayford Steele
    B. Harvey Minefield
    C. Peter Schraut
    D. Emanuel Schleussinger

8. What is the name of the Game Boy Advance emulator used in this book?
    A. DevKit-Advance
    B. BoycottAdvance
    C. VisualBoyAdvance
    D. Big Fat Emulator

9. What is the URL for the HAM SDK Web site?

    A. http://www.ngine.de

    B. http://www.console-dev.org

    C. http://vboy.emuhq.com

    D. http://www.nintendo.com

10. What is the name of the device that reads and writes Game Boy Advance flash cartridges?

    A. Multi-Boot Version 2

    B. Game Boy Flasher

    C. Cartridge Flash Linker

    D. Flash Advance Linker

Chapter 4

Starting With

The Basics

This chapter builds upon the previous one by continuing to increase your familiarity with the Visual HAM environment. It does so by developing several sample programs from scratch, showing you how to compile and run them. These programs are simple in nature and are not meant to focus on any particular subject (such as high-speed graphics, covered in later chapters). Instead, this chapter focuses on writing quick and simple programs using Hamlib and stock GBA code. If you are already familiar with GBA coding and you desire to get directly into graphics programming, you may skip this chapter and move on to the next one. The entire second part of the book, "Being One With The Pixel," is, as the name suggests, dedicated solely to graphics programming.

This chapter shows you how to write several complete programs from start to finish, so you will be prepared for the rest of the material in the book. The Visual HAM environment is capable of compiling any GBA program, but the HAM distribution comes with an excellent GBA wrapper library called Hamlib, which abstracts much of the low-level memory addressing code with actual function calls and callbacks that are just more intuitive and especially helpful for those who are new to console programming. The focus of this chapter is also to provide you with some experience writing, compiling, and running programs on either the emulator, multiboot, or flash linker and is therefore helpful for increasing your logistical skills with the development environment.

Specifically, this chapter covers the following subjects:

- The basics of a Game Boy program
- Displaying a friendly greeting
- Drawing pixels
- Filling the screen
- Detecting button presses
- Running programs directly on the GBA

The Basics Of A Game Boy Program

Programmers are impatient people. We want to see something happen as quickly as possible, even if it's not realistically humanly possible to do so. The motto of the programmer is often "Make it work, then fix it." Unfortunately, most of us love to write code but don't particularly like to design things. After all, it's far more fun to get started with hammer and nails rather than pencil and paper, right? This chapter is not about game design, although it is a subject that permeates the book, because design is part of the whole development process. The Game Boy Advance is a very simple computer, in the sense that it has a processor, has memory, and can run programs. The programs are meant for entertainment and don't always take the form of a game. There is a printer available for the Game Boy, for instance, and a digital camera that allows one to take photos and print them out. This is obviously not even remotely related to playing games, but it is fun nonetheless. Plus, you have to admit that it's cool being able to take pictures and print them out on a small handheld video game console!

Using the tools provided with this book and the knowledge gained from reading its contents, you have an opportunity to write any program you want for your own edification or entertainment. Many aftermarket accessories and programs are becoming available for GBA owners. For instance, there is an aftermarket MP3 player available for GBA! While I would debate the usefulness of such an accessory, there are many who would enjoy using their GBA to play music in addition to playing games. Given the versatility of the GBA platform, it's no wonder such things are commercially viable products. For example, one aftermarket accessory I purchased was the Afterburner backlight kit for my own personal GBA. I had a difficult time with the dark stock screen that is prone to light glare and simply could not deal with it while developing on it. Of course, at the present time, one can now just purchase a GBA SP with a built-in backlight and rechargeable batteries.

What Makes It Tick?

But I digress, the subject at hand is this: What makes a Game Boy Advance program tick? In a nutshell, a GBA program is just a binary ROM image that the hardware runs as if it were an integral part of the system. In other words, game cartridges fool the GBA into thinking the game was always there, because it isn't smart enough to know that you have removed the previous game and inserted a new one. But essentially, the GBA functions by assuming that a cartridge is a permanent piece of hardware. Let's dig into this line of thinking with more detail in the next section. From a programming perspective, how the GBA reads a cartridge

is not as important at this point as knowing how to write GBA code in the first place, so I won't get into that extensively. You have already learned a great deal about the GBA hardware and how the console works from the last few chapters.

In many respects, these more difficult issues are what make Hamlib so appealing. When you consider that this library is a complete GBA framework capable of being used to produce commercial games, there is no reason to discount it as yet another wrapper library (a common complaint by software gurus). Hamlib is an awesome library that I will introduce to you in this chapter. But I recognize the valid point many programmers make in that they want to be as close to the hardware as possible—for many obvious reasons, such as knowing exactly what is going on, writing the fastest code possible, and so on. I share those sentiments, but I am also very encouraged by the strengths of Hamlib and the work put into the frequent updates to the library (as well as frequent updates to the HAM distribution in general).

So, how about if we abstract the hardware side of things from this point forward and focus on software development? If you are a programmer first (like I am) and ■ hardware hobbyist second, then the hardware is only of passing interest—which is most likely limited to knowing how a flash linker works, and so on. Now, on to what we programmers do best—writing source code.

A Friendly Greeting

How about a practical and easy-to-understand sample program to get things rollinling? There's no better teacher than direct experience. You have already seen your first aftermarket GBA program running (that is, an unlicensed GBA game), the ShowPicture program from the previous chapter. I'm not going to open that project again, but you are free to return to the ShowPicture source code at any time as you increase your GBA coding skills (and you will explore bitmapped graphics extensively in Part Two). In the meantime, let's write ■ simple program that displays a message on the screen.

The important thing that I want you to grasp in this chapter is not the programming language used or the usefulness of a particular program, but rather just getting a good feel for how the development environment works; how to write code and compile your programs; and how to test programs using the emulator, flash linker, or multiboot cable. Therefore, I will start with a very simple program that just displays a message on the GBA screen. This first sample program that you will write uses the HAM library (Hamlib) to

display text on the screen. Normally, something as seemingly simple as displaying a message would require a lot of work up front, because the GBA has no built-in function for drawing text on the screen. That's right, something as simple as a message requires a lot of work, which often involves creating a bitmap filled with font characters (or byte array of hard-coded letters) and writing a function to display a message using the font. It's all such an enormous amount of work for something so simple!

That is the way of things in console development—you have to do everything yourself. Even something as simple as polling a timer in order to maintain a constant refresh rate in a game is a very low-level activity, requiring intimate knowledge of how timers and interrupts work (see chapter 8, "Using Interrupts And Timers"). Now let's get started on the Greeting program.

Creating A New Project

If you haven't already, go ahead and fire up Visual HAM by double-clicking on the icon I helped you to create on your desktop or by browsing to the HAM installation folder and searching for VHAM.EXE. If you installed Visual HAM to the folder that I recommended, that might be located in C:\HAM\VHAM. Next, open the File menu and select New, then New Project, as shown in Figure 4.1.

Figure 4.1

Creating a new project from the File menu of Visual HAM.
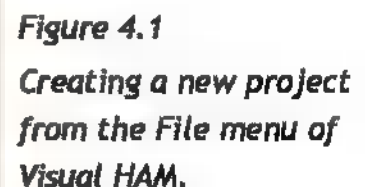
The New Project dialog box should appear, as shown in Figure 4.2. Select the first project type, [C] Empty, and then type "Greeting" for the project name. Select an appropriate location for this project on your hard disk drive. Click on the OK button to create the new project, and open the editor on the default source file.
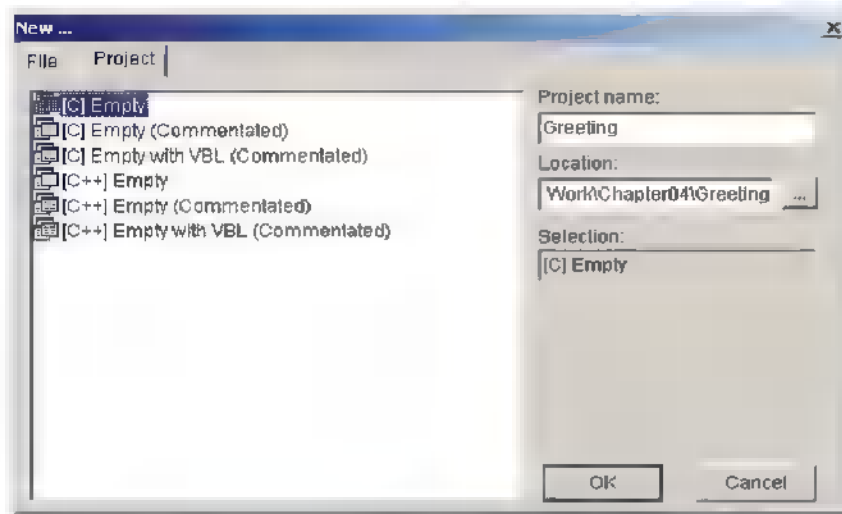
Figure 4.2
The New Project dialog box is where you specify the name, location, and type of project.

The source file for a new project in Visual HAM looks like the file shown in Figure 4.3. The default skeleton source code was already added to the main.c file for you by Visual HAM.
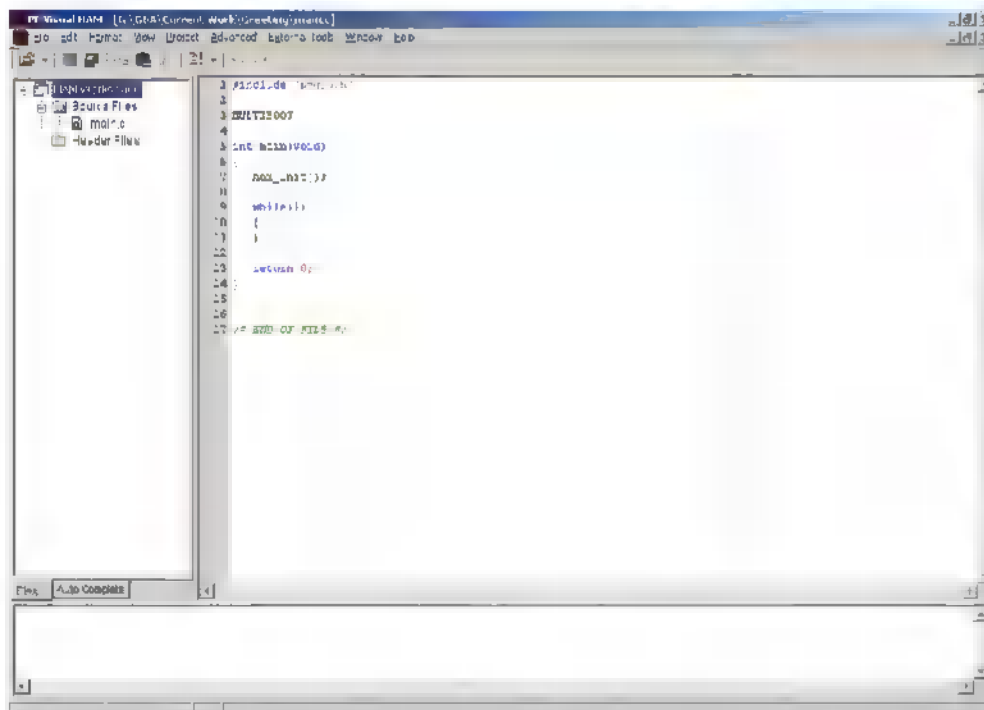
Figure 4.3
A new project in Visual HAM with generated skeleton source code.

The source code looks like this:

```
#include "mygba.h"


MULTIBOOT


int main(void)
{
    ham_Init();
    while(1)
    {
    }


    return 0;
}


/* END OF FILE */
```

This default code is the minimum amount of code needed for a HAM program, and it actually *will* run (although it doesn't do anything useful). While the Greeting program you are about to write is indeed a HAM program (meaning that it uses the HAM library), there are many ways to write a GBA program, such as with straight C or C++, without any library at all. I will show you how to write a raw GBA program in the next section of this chapter (the next sample project, in fact). One interesting line is the MULTIBOOT statement. That is a specific statement that the GBA compiler recognizes and is somewhat like a macro of a define. It is needed when using the multiboot cable (more on that later in this chapter).

Writing The Greeting Program Source Code

Okay, let's modify the skeleton program that Visual HAM generated for the Greeting project. There are two sections of code that I would like you to add to the program, as indicated in the code listing that follows. The new code is denoted in bold font. First, a line of code that initializes the Hamlib text system, and then three lines of code inside the while loop to actually display a message on the screen. Go ahead and modify the listing now as shown.

```
#include "mygba.h"

MULTIBOOT

int main(void)
{
    ham_Init();

    //initialize the text system
    ham_InitText(0);

    while(1)
    {
        //display a greeting message
        ham_DrawText(0, 0, "Greetings!");
        ham_DrawText(0, 2, "Welcome to the world of");
        ham_DrawText(0, 4, "Game Boy Advance programming!");
    }

    return 0;
}


/* END OF FILE */
```

Not bad, is it? The program is very short, and you should be able to easily understand what the program will do. It uses the ham_DrawText function to display a text string on the screen. Note that this function only works if you have first called ham_InitText, because that function loads the bitmap font used by the ham_DrawText function. It displays a nice system-type monospaced font that looks as if it were part of a built-in GBA text library, although you now know that Hamlib actually does all the work there!

Compiling The Greeting Program

Now let's compile the program. I will also show you some shortcut keys that you can use in Visual HAM to compile and run programs. If you look at Figure 4.4, it shows the Project

menu in Visual HAM. Open the Project menu now, and select the Build menu item. This will invoke the compile process.

Figure 4.4
The Project menu in Visual HAM, showing the Build menu item.

If the program has no syntax errors or typos, you should see no error messages appear in the output window down at the bottom of the screen in Visual HAM. When there is an error, it will be highlighted with a red "ERROR" message, which also displays the line number where the error occurred. I will get into debugging and error handling in later chapters. For now, if you see any error messages, the problem is most likely a typo, which you should be able to resolve by comparing the listing shown here with the source code on your screen. If there seems to be an error resulting from something other than ■ typo, it is possible that your installation of HAM is damaged, and you may want to refer back to the previous chapter to perform a reinstall of HAM. A common source of errors is when there are two different versions of HAM installed on your PC at the same time. Be sure to delete any older version before installing the latest version of HAM (which may be updated from the version included on the CD-ROM).

Press F5 to compile a project in Visual HAM.

Testing The Greeting Program In The Emulator

You are now ready to run the Greeting program on your PC using the emulator included with HAM, ■ program called VisualBoyAdvance. If you open the Project menu once again, look for the menu item called Build + VBA (as shown in Figure 4.5), and select it. The program

should compile and begin
running in the emulator, as
shown in Figure 4.6.

Press F7 to compile the project and run it in the emulator.

Figure 4.5
The Project menu,
showing the Build +
VBA menu item.

Figure 4.6
The Greeting
program running in
the emulator.
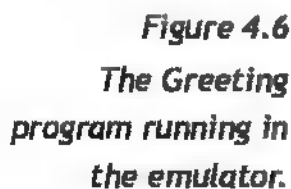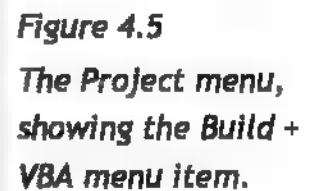
In this screen shot, you can see that I have increased the default window size of VisualBoyAdvance to two times the normal size. You can change the size of the emulator window yourself by opening the Options menu (as shown in Figure 4.7), selecting Video, and then choosing a screen size for the emulator, from 1x up to 4x, and even one of several full-screen modes (which I don't recommend during development but encourage you to at least try out).
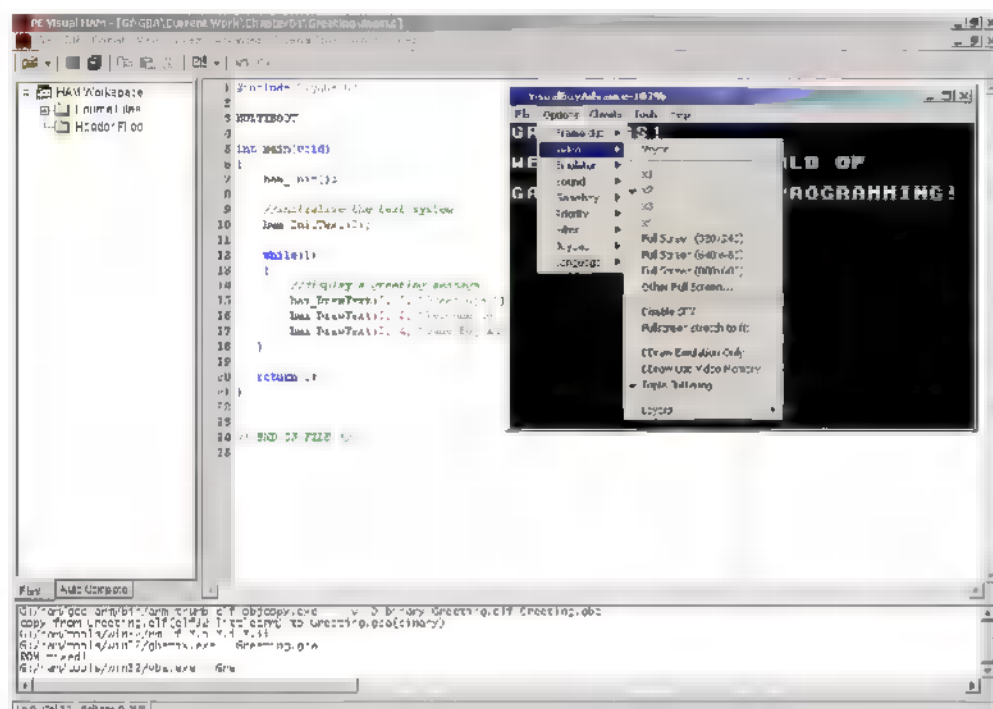
Figure 4.7
Changing the window size of the emulator using the Options menu.

Drawing Pixels

Now for something really fun! This program might surprise you. After all, it's a real GBA program, and yes, it will run on an actual GBA (using a flash linker, for instance). The great thing about this program is just how small it is. Now, I realize this doesn't do much, but it's ■ 100 percent bona fide GBA program, and it does one of the most basic things that you must learn when programming video games—drawing pixels. This is the basis for all video games! Drawing a single pixel in video game terms is sort of like the Hello World mantra established by Brian W. Kernighan and Dennis M. Ritchie in their famous book *The C Programming Language* (the book that first introduced the world to the C language—the language used in this book). The Greeting program you just wrote was a sort of Hello World program, but it used Hamlib (because, as I explained, it would be too difficult to display text without Hamlib at this point).

Now, assuming Visual HAM is still running from the last project, what you will want to do is create ■ new project. Alternatively, you may load the project from the CD-ROM (\Sources\Chapter04\); you can do this for any of the projects in this chapter. But where's the fun in that? This program is little, so I insist you type it in! However, for future reference, note that Visual HAM project files have an extension of .vhw. So, anytime you need to open a GBA project from within Windows Explorer, you can simply double-click the .vhw file (in this case, DrawPixel.vhw). There is usually also a binary executable file with an extension of .gba along with each project. Double-clicking the .gba file should cause the emulator to start. If it doesn't, simply locate the VisualBoyAdvance.exe (or vba.exe) program file on your hard drive in order to associate .gba files with the emulator.

Create a new blank project and delete whatever default code Visual HAM fills in automatically. We're going to start from scratch here. Name the project DrawPixel and give it a new project folder (which is created by Visual HAM). If you are having trouble creating the project, refer to the figures from the previous sample program to refresh your memory.

Writing the DrawPixel Program Source Code

Let me first explain this code a little. I left it intentionally sparse in order to make a point, that a GBA program can be very small, and small it is indeed! I'm not sure if it's possible to write ■ smaller GBA program than this (excluding the comments, of course). It is really only . . . let's see . . . eight lines of code, counting the curly brackets. Okay, go ahead and type it in. I'll explain what is going on in this program after the listing.

```
//////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// DrawPixel Project
// main.c source code file
//////////////////////////////////////////////

int main(void)
{
    // create a pointer to the video buffer
    unsigned short* videoBuffer = (unsigned short*)0x6000000;
```

```
// switch to video mode 3 (240x160 16-bit)
// by setting a memory register to a specific value
*(unsigned long*)0x4000000 = (0x3 | 0x400);


// draw a white pixel (16 bits) directly to video memory
// pixel location is centered on the screen (120,80)
videoBuffer[80 * 240 + 120] = 0xFFFF;


// continuous loop
while(1) { }


// end program
return 0;
}
```

After you have finished typing in the code, the IDE should look like Figure 4.8.
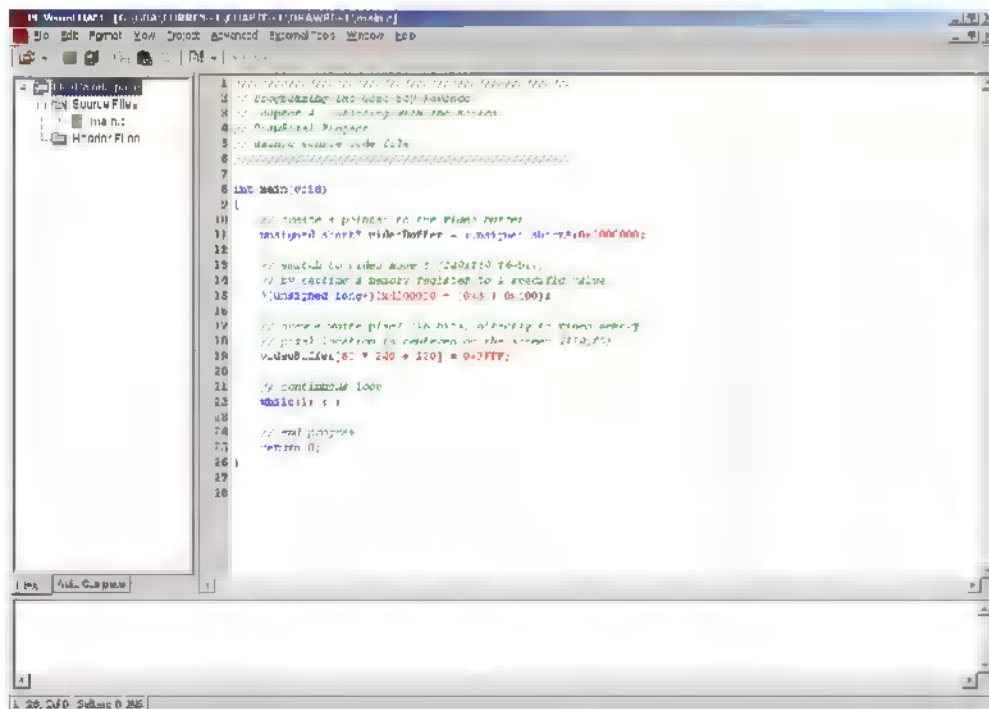
Figure 4.8
The DrawPixel
project as it
appears in the
Visual HAM IDE.

This program jumps ahead a bit, because graphics are really covered in Part II, starting with the next chapter. But I wanted to give you a taste of what is to come. This is a short program, but it provides a basis for just about any GBA game you are likely to write in the future.

Are you surprised to find no includes in this program? If you are an old hand with C, you are likely wondering why there are no header files for interfacing with the GBA. That's the beauty of the HAM distribution and the Visual HAM IDE. There are really no headers included by default, and none are needed for the most basic programs, although the HAM SDK includes several libraries automatically when the program is compiled and linked into an exe file. You will face this situation throughout the book. The GBA uses memory registers to perform basic functions. For instance, the code that sets the video mode:

```
*(unsigned long*)0x4000000 = (0x3 | 0x400);
```

is just a pointer to a memory location, and a specific numeric value is set in that memory location. In this instance, 0x3 is video mode 3: 240 x 160, 16-bit, while 0x400 refers to background 2, and these values are combined with a bitwise OR (the pipe symbol, |). I will explain these things in more detail in Part Two, which is dedicated entirely to graphics programming. For now, the point is to get a feel for *what* a GBA program is like, rather than specifically *how* every line of code works.

The next non-comment line of code:

```
videoBuffer[80 * 240 + 120] = 0xFFFF;
```

actually draws the pixel at the center of the screen. Video mode 3 has a resolution of 240 x 160, so the center of the screen is at 120 x 80. The formula for accessing a linear memory array using two-dimensional coordinates (in this case, the pixel's X,Y location) is this:

```
Memory Location = Y * Screen Width + X
```

By filling in this memory location formula for the location in the video buffer, you are able to then set that memory location to a specific value—the color of the pixel, which was set to 0xFFFF in this case. 0xFFFF is a hexadecimal number, where each character after the 0x is 4 bits in size (with possible values of 0-9 and A-F, for a total of 16 bits). Therefore, 0xFFFF is ■ 16-bit number, which is exactly what is needed for mode 3, because it uses 16-bit pixels. In the graphics chapters of Part II, I will explain how to set the pixel color using the usual (Red,Green,Blue) components—which is somewhat beyond the scope of this short example.

Compiling the DrawPixel Program

Now, I'm sure you have already jumped ahead and run the program, per the instructions provided in the previous sample program. If you have not already done so, go ahead and compile (or rather, build) the program by pressing F5. If all goes well and there are no syntax errors in your program, then it is ready to be run in the emulator.

Testing the DrawPixel Program in the Emulator

At this point, you may open the Project menu and select Build + VBA to run the program (or simply press F7 to perform this step with a single keystroke). The running DrawPixel program is shown in Figure 4.9. Do you see the small pixel in the center of the emulator window?
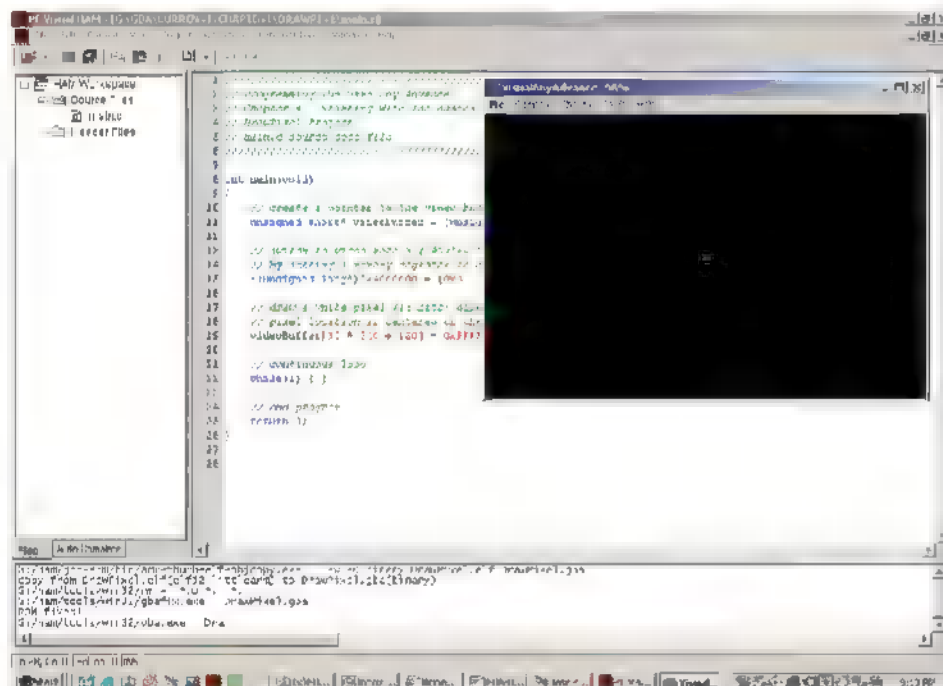
Figure 4.9

The DrawPixel program draws ■ *single pixel in the center of the GBA screen.*

Filling The Screen

Now, after seeing just one pixel on the screen, I can't resist the temptation to take it ■ step further and fill the entire screen with pixels! This code is just starting to become fun to write, so let's go for it and write another graphics program. I know, this is all covered in Part Two, as I have explained already, but it couldn't hurt to take a peek ■ bit early, right?

Writing the FillScreen Program Source Code

You should be pretty good at creating a new project in Visual HAM after the last two sample programs, so I won't go over that again right now (although I will go through the process at least once in each chapter, lest you get lost at any point). This program, which is called FillScreen, uses some defines and one function, simply because it's too hard to remember the special memory addresses of the GBA from memory (how's that for a tongue twister?). Advanced GBA projects use an include file with all of the memory addresses and registers in the GBA (see Appendix C, "Game Boy Advance Hardware Reference"), so I might as well give you a sneak peek at what some of those things look like. This program creates ∎ define for the memory register for changing video modes, which you saw in the previous sample program. The register is actually called REG_DISPCNT, and the define looks like this:

```
#define REG_DISPCNT *(unsigned long*)0x4000000
```

In case you aren't a C guru (and I'm not expecting you to be, although I know some folks out there are old hands with C), the #define statement allows you to create ∎ macro that the compiler fills in at compile time. What this means is that anytime the compiler finds REG_DISPCNT in the source code, it fills in *(unsigned long*)0x4000000 in its place! There's no denying that this little feature will preserve your sanity, because there are many, many memory registers in the GBA architecture that resemble this particular one. What happens, specifically, is that when the GBA detects a change at that memory address, it knows that it should change video modes to the number specified. This is very much like a function call, as if there is a sort of SetMode function built into the GBA. While it isn't called SetMode, the memory register is essentially the same thing. When you set the memory register to a specific value, such as 0x3, you are actually passing a parameter to the "function", so to speak. Are you following me? If not, that's okay, because I'll explain each new memory register as needed in later chapters, so you'll get the hang of it in time. I will admit, this is ∎ new and foreign way to write code, especially for those of us who are used to procedural or object-oriented programming. But that is what makes console coding so rewarding—you are closer to the hardware and actually manipulating physical parts of the memory built into the GBA, in order to do things.

There are three more defines in this program as well. The next one:

```
#define MODE_3 0x3
```

you may recognize from the DrawPixel program. This is the video mode that the program uses, mode 3, with a resolution of 240 x 160 and 16-bit color depth. By defining it to MODE_3, it's easier to remember exactly what mode the program is using. I realize that 0x3 is easy to spot as well, so if you prefer that, go ahead and use the literal instead of the define (that is what I do in later chapters).

The next define:

```
#define BG2_ENABLE 0x400
```

is also related to the video mode. As you may recall, the DrawPixel program set the video mode by OR'ing 0x3 with 0x400, in order to specify that the program should use background 2. This is something that I will cover, again, in Part Two, so don't worry about it at this point.

The last define:

```
#define RGB(r,g,b) (unsigned short)(r + (g << 5) + (b << 10))
```

creates a macro for packing an RGB color into a 16-bit value. This allows you to pass parameters to the define, as if it were a function. In fact, this could be written as a function instead of as a define, but the define is simpler.

Finally, the DrawPixel3 function:

```
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}
```

This function, as the name implies, draws a pixel on the mode 3 screen. The single line of code in this function also resembles the code in the DrawPixel program, but this version now allows you to pass the X,Y values as parameters. It is, therefore, more useful as a function. Now here is the complete source code listing:

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4 - Starting With The Basics
// FillScreen Project
```

```c
// main.c source code file
/////////////////////////////////////////////////


//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000


//video mode 3 = 240x160 16-bit
#define MODE_3 0x3


//use background 2
#define BG2_ENABLE 0x400


//macro to pack an RGB color into 16 bits
#define RGB(r,g,b) (unsigned short)(r + (g << 5) + (b << 10))


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//draw a pixel on the mode 3 video buffer
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}


/////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////
int main(void)
{
    int x, y;

    //switch to video mode 3 (240x160 16-bit)
    REG_DISPCNT = (MODE_3 | BG2_ENABLE);
```

```
//fill the screen
for (x = 0; x < 239; x++)
{
    for (y = 0; y < 159; y++)
    {
        DrawPixel3(x, y, RGB(0,(255-y),x));
    }
}


// continuous loop
while(1)
{
    // do nothing
}


// end program
return 0;
}
```

The key to this program is the section of code denoted by the comment "fill the screen".
Here are two for loops: the first for the X values, the second for the Y values. Inside the
loops is a call to DrawPixel3 with the X and Y variables. A creative use of the Y value
provides the color, thus filling in the screen with an interesting fill effect.

Compiling the FillScreen Program

Now, go ahead and compile the program by pressing F5. This is the most complicated
program you have written so far, so don't be surprised if there are a few syntax errors. The
most common errors involve a missing semicolon at the end of a line or a missing closing
curly brace at the end of a block of code (such as with a for loop). If there are any errors,
closely examine the source code on your screen and compare it with the printed listing to
locate errors. Another potential source of errors is the case of variable names. Remember
that in C, everything is case sensitive, so that X is recognized as a different variable than x.
This can be very confusing at times, so take care to watch the case when naming variables.

Testing the FillScreen Program in the Emulator

Now, one of the reasons why I include the compile step separately from the testing, or running, step here is to make sure the program works first. Obviously, after you are more experienced with Visual HAM and have been working on a program for hours, you will likely just skip the compile step and go directly to the Build + VBA step by pressing F7. This is what I usually do after the first few times. I often first compile a program when loading up someone *else*'s game (because there are a lot of public domain GBA games available online—see Appendix B for a list of good Web sites featuring fan-written GBA games). HAM is capable of running programs not even created under Visual HAM, because it uses the same GCC compiler that the other GBA development kits use. However, that may not always be the case, as new development kits are appearing all the time; such is the case in the open source community. Usually, though, most programmers use the top one or two development kits, and HAM is definitely one of those.

Go ahead and run the program now. If all goes well, you should see the emulator window appear with ■ colorful pattern filling the simulated GBA screen, as shown in Figure 4.10.
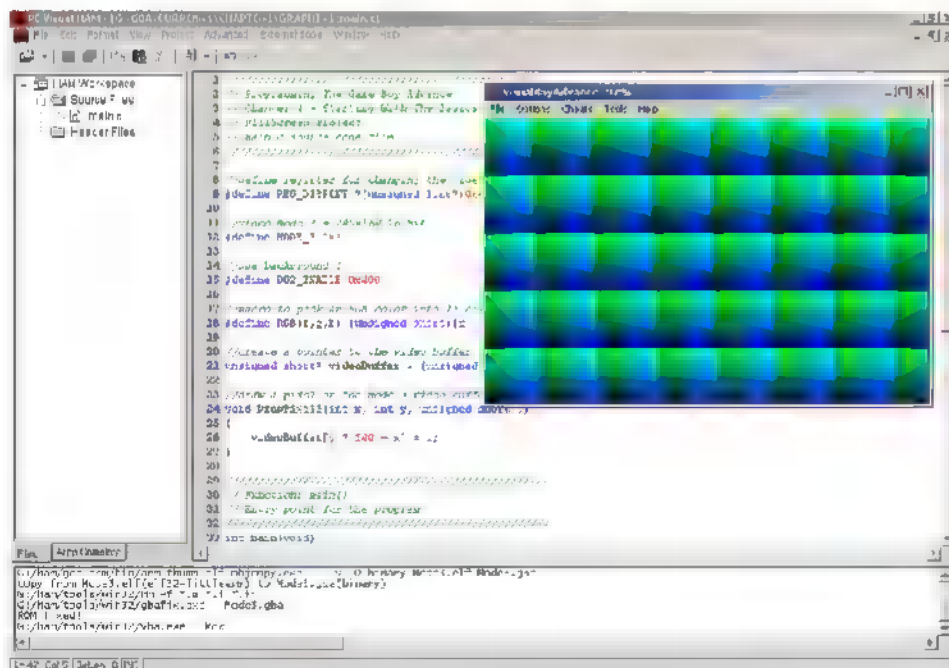
Figure 4.10

The FillScreen program fills the GBA screen with a pattern of pixels.

You know, with the source code for FillScreen, you have everything you need to write ■ rudimentary GBA game at this point. That is amazing considering that you were only drawing your first pixel a short while ago. But the fact remains that once you are able to draw a pixel, it follows that you are able to write a game with only ■ little more effort.

Such a game might not have advanced, high-speed blitting (a fast method of drawing graphic images) or transparency, but it is still a significant possibility.

But wait, something is missing. First, you need to be able to capture button presses! At present, all you can really do is write a demo—something interesting graphically, but with no possibility for input. <Sigh>. Okay, there's still a lot of ground to cover before writing your first game, but I don't want to discourage you. Therefore, let's take a quick tutorial on reading button presses on the GBA.

If you are at all excited about these basics, wait until you get to Chapter 7, "Rounding Up Sprites," where you'll learn how to use hardware-accelerated sprites with built-in transparency, alpha-blending, rotation, and scaling capabilities! Not only that, in Chapter 6, "Tile-Based Video Modes" will teach you how to create scrolling backgrounds. By the time you have finished those chapters, you will have no need for pixels at all. But it's nice to start with the basics, because that helps ones to appreciate what the GBA can do.

Detecting Button Presses

This section includes a program called ButtonTest that—surprise!—detects the GBA buttons. Because this is such a significant part of gameplay, and the subject isn't covered fully until Chapter 10, "Interfacing With the Buttons," I wanted to give you a little exposure to button programming at this point. I know you will have some fun with the code presented in this program! The ButtonTest program uses the ham_DrawText function to display text messages on the screen in order to update the status of each button, which appears by name on the screen (with a small "x"). The button presses are detected by a series of if . . . else statements and button macros such as F_CTRLINPUT_UP_PRESSED (which is specific to Hamlib).

Writing the ButtonTest Program Source Code

The source code for the ButtonTest program is about a page and a half in length, and some of it comprises comments (which you may leave out, if you wish). Basically, this program is a simple loop with a continuous conditional check of the buttons on the GBA using a series of if . . . else statements. First, the program initializes HAM by calling ham_Init, which must be called before using any of the features of the HAM library. Next, ham_InitText(0) is called to initialize the text mode of Hamlib using a specified background number (I will talk

more about backgrounds in Part Two). After these two initializing functions have been called, the program uses ham_DrawText to display the status of each button on the screen.

First, you need to create a new project called ButtonTest, using the same procedure you have followed for the last three sample programs. Fire up Visual HAM, open the File menu, and select New, and then New Project to open the New Project dialog box, as shown in Figure 4.11. Select the [C] Empty project type. For the project name, type in "ButtonTest", and then type in the folder where you would like the project to be created. Note that Visual HAM will create the folder if it doesn't already exist.
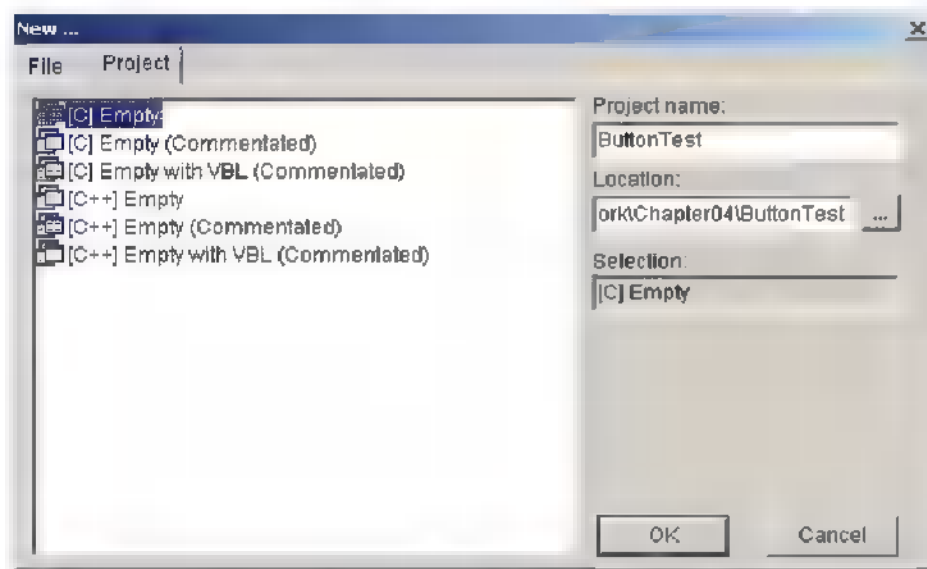
Figure 4.11
The New Project
dialog box.

Next, just delete the skeleton code Visual HAM generated for you, and type in the following code listing for the ButtonTest program. Or, if you are good at filling in the details, you may simply type this program into the skeleton code, filling in where necessary, because the generated code is included in this listing. Just be sure not to leave out anything, as it's easy to lose your place while filling in code (just as it's easy to lose your place when typing in an entire code listing from scratch).

I emphasize typing because there truly is no better way to familiarize yourself with a new programming language or SDK. If you simply load up each of the sample programs from the CD-ROM, you may run them and see what the programs look like. However, you lose that critical step—typing in the code makes you intimately familiar with the function calls and gives you deeper insight into how the program works.

Here is the complete listing for the ButtonTest program:

```
///////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// ButtonTest Project
// main.c source code file
///////////////////////////////////////////////


// include the main ham library
#include "mygba.h"


// enable multi-boot support
MULTIBOOT


///////////////////////////////////////////////
// Function: main()
// Entry point for the program
///////////////////////////////////////////////
int main()
{
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
```

```
ham_DrawText(3,9,"R");

ham_DrawText(3,10,"START");

ham_DrawText(3,11,"SELECT");


// continuous loop

while(1)

{

    // check UP button

    if (F_CTRLINPUT_UP_PRESSED)

        ham_DrawText(0,2,"X");

    else

        ham_DrawText(0,2," ");


    // check DOWN button

    if (F_CTRLINPUT_DOWN_PRESSED)

        ham_DrawText(0,3,"X");

    else

        ham_DrawText(0,3," ");


    // check LEFT button

    if (F_CTRLINPUT_LEFT_PRESSED)

        ham_DrawText(0,4,"X");

    else

        ham_DrawText(0,4," ");


    // check RIGHT button

    if (F_CTRLINPUT_RIGHT_PRESSED)

        ham_DrawText(0,5,"X");

    else

        ham_DrawText(0,5," ");


    // check A button

    if (F_CTRLINPUT_A_PRESSED)

        ham_DrawText(0,6,"X");
```

```
    else
        ham_DrawText(0,6," ");


    // check B button
    if (F_CTRLINPUT_B_PRESSED)
        ham_DrawText(0,7,"X");
    else
        ham_DrawText(0,7," ");


    // check L button
    if (F_CTRLINPUT_L_PRESSED)
        ham_DrawText(0,8,"X");
    else
        ham_DrawText(0,8," ");


    // check R button
    if (F_CTRLINPUT_R_PRESSED)
        ham_DrawText(0,9,"X");
    else
        ham_DrawText(0,9," ");


    // check START button
    if (F_CTRLINPUT_START_PRESSED)
        ham_DrawText(0,10,"X");
    else
        ham_DrawText(0,10," ");


    // check SELECT button
    if (F_CTRLINPUT_SELECT_PRESSED)
        ham_DrawText(0,11,"X");
    else
        ham_DrawText(0,11," ");
}
```

```
// end program

return 0;
}
```

After you have finished typing in the source code for the ButtonTest program, the editor should look something like Figure 4.12.
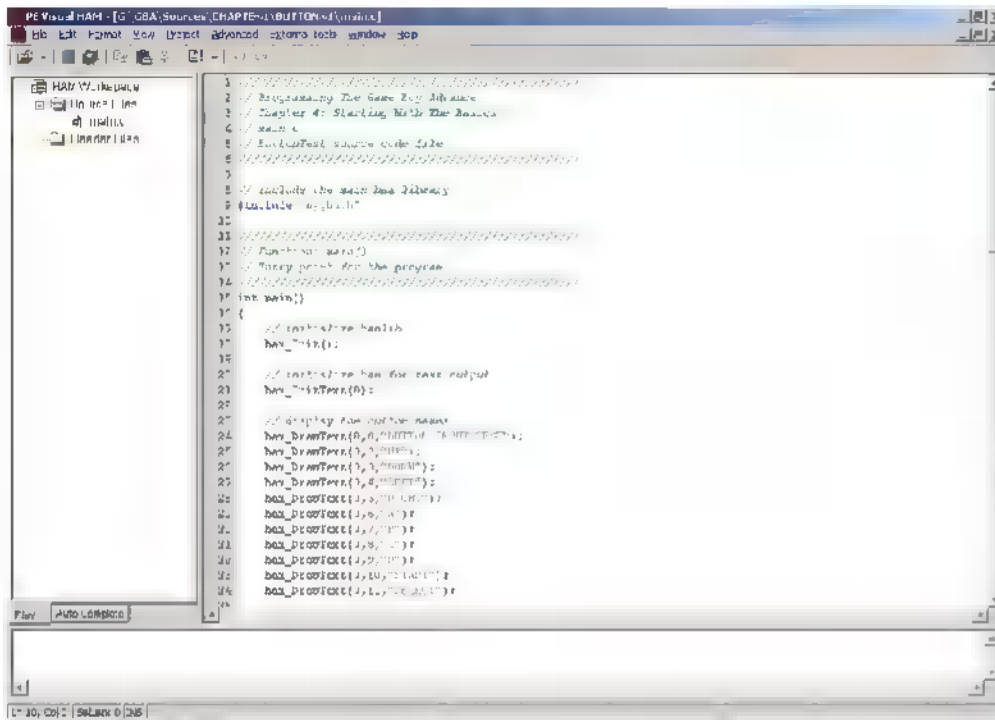
Figure 4.12
The ButtonTest program demon-strates the button handler built into Hamlib.

Compiling the ButtonTest Program

Now let's compile the ButtonTest program. Just for reference, because it's been a while, I'll go through the steps with you again. First, open the Project menu and select Build, as shown in Figure 4.13.

Now for a little more detail as to what is happening. The build command invokes the make utility to run the makefile that is generated by Visual HAM when you created the project. After invoking a build, the IDE looks like Figure 4.14. Note the compiler messages at the bottom of the screen.
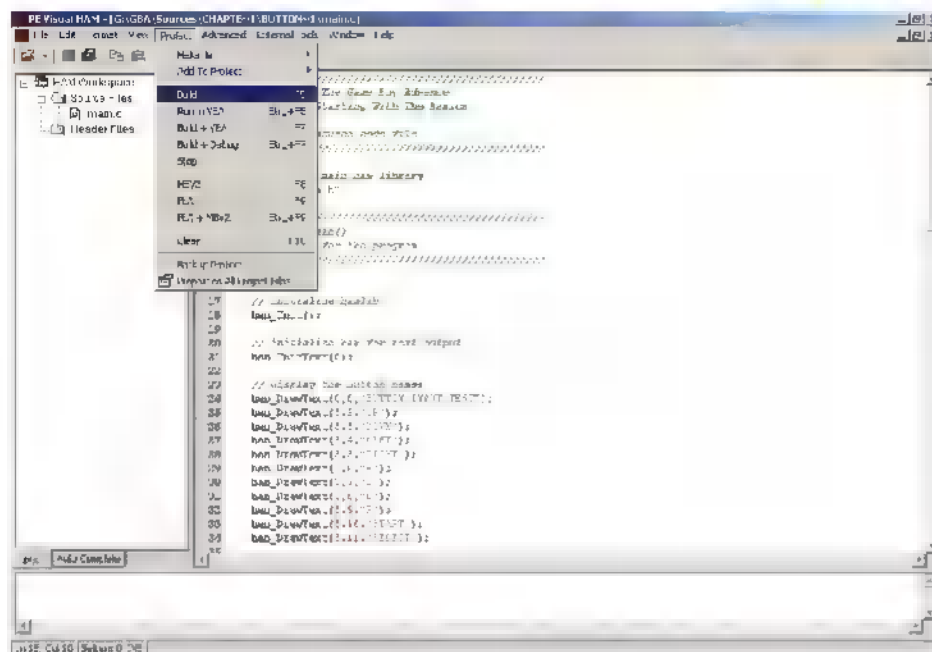
Figure 4.13
Use the Project, Build menu to compile a program in Visual HAM.
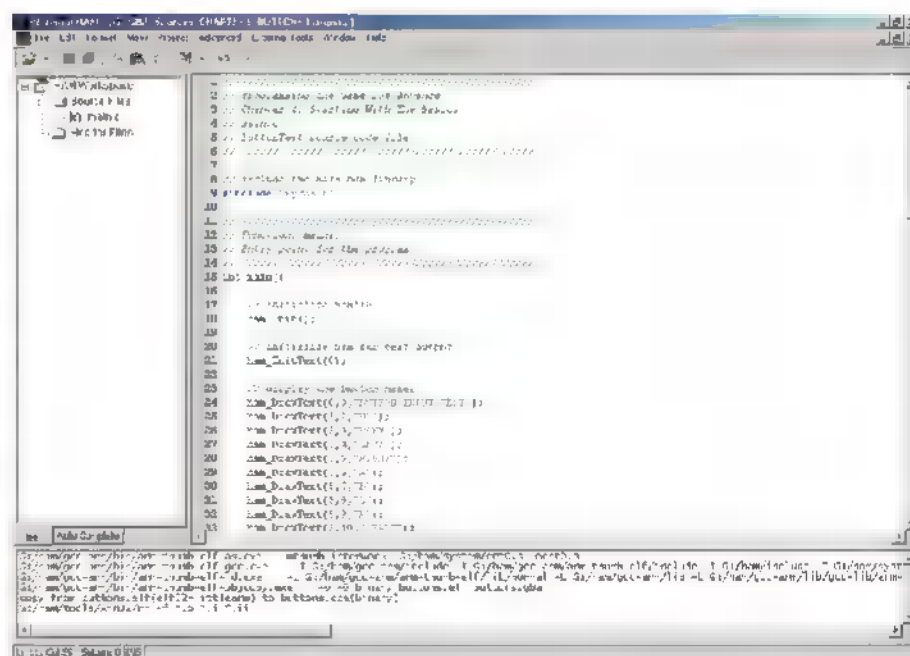
Figure 4.14
After compiling the project, the status window at the bottom of Visual HAM displays messages from the compiler.

Testing the ButtonTest Program in the Emulator

One last time, I'll go over the run process, just to make sure you've got it down. VisualBoyAdvance is the GBA emulator that comes with HAM and may be invoked from the IDE by opening the Project menu and selecting Build + VBA, as shown in Figure 4.15, or by simply pressing the F7 key.
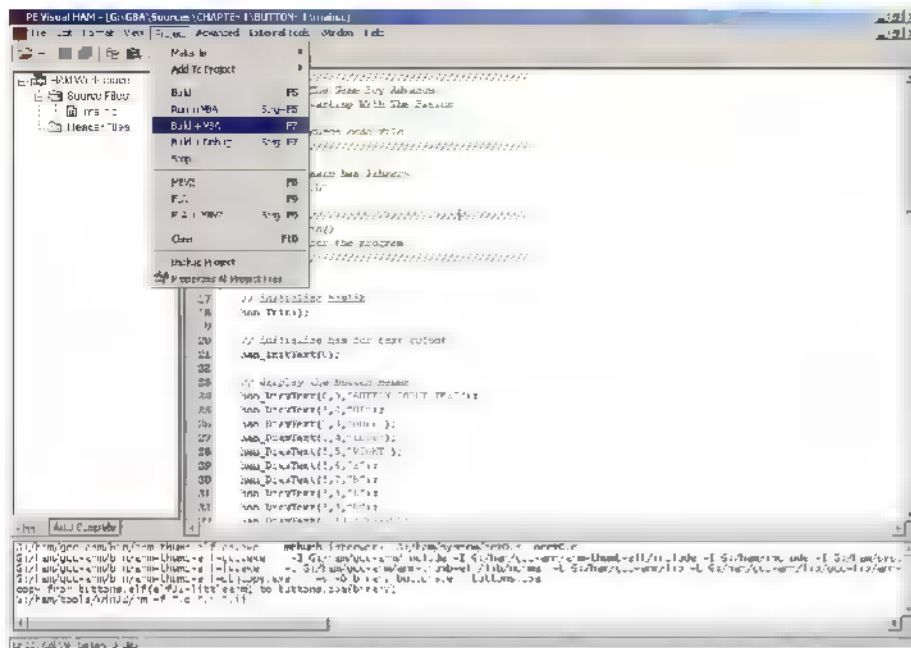
Figure 4.15

The Build + VBA menu item will compile a program and run it in the emulator.

When you do this, Visual HAM will start the compile process. If the program compiles without errors, the program is run in the emulator, which then appears on the screen, as shown in Figure 4.16.

Figure 4.16

The ButtonTest program has been compiled and is shown running in the emulator.
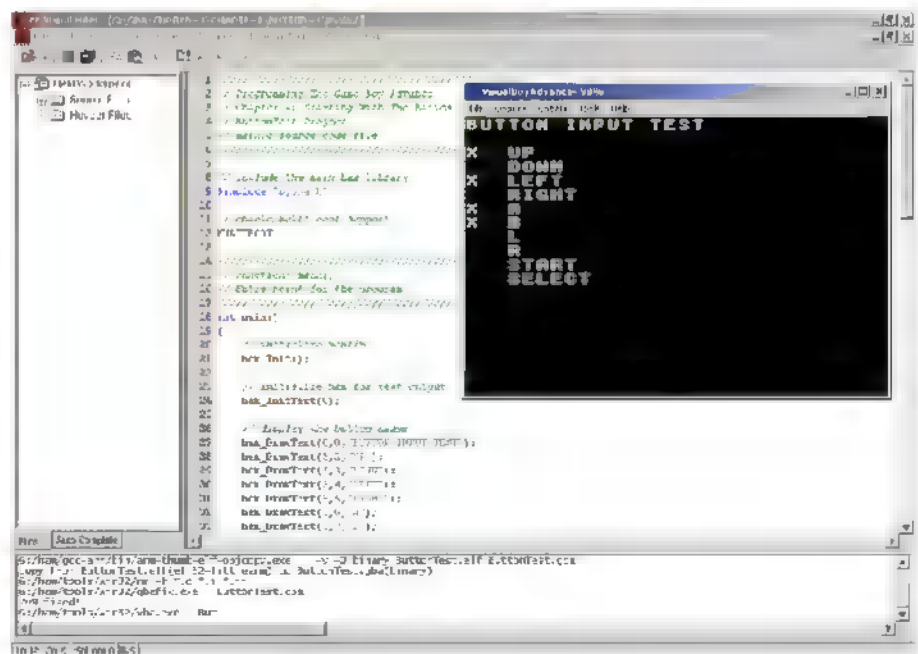
Running Programs Directly on the GBA

The Game Boy Advance is a portable video game machine, so sooner or later you will want to take your games and demos with you or at least be able to run the programs on the

actual GBA—to see how well it plays on the real hardware. There are two ways to do this. First, there is the Multi Boot Version 2 (MBV2) device, which connects the GBA to your PC using ■ parallel port adapter. Then there is the Flash Advance Linker (of which there are many different models and types), which can read and write to flash cartridges. If you end up purchasing one of these hardware devices, you will get instructions on how to use it.

Since there are so many devices available, I won't go into detail about specifically what steps to take to install the driver (if there is one) or how to use the software. When you connect either your MBV2 or Flash Advance Linker to your PC and are able to successfully use it as described in the product's enclosed instructions (which may be printed or may be in electronic form on an enclosed floppy disk or CD-ROM), then you will be able to use it with Visual HAM. The process is fairly easy from that point forward. The initial installation and testing require more effort than actually using one of these devices, but the reward—of being able to see your programs running on a real GBA—are definitely worth the cost and effort of acquiring and installing one.

The Multiboot Cable

The multiboot device (formally known as the Multi Boot Version 2, or MBV2) allows you to run programs directly on the GBA without a flash linker by taking advantage of the GBA's multiplayer capabilities. When a GBA detects a multiplayer cable inserted into the Ext port, it will attempt to download a small binary program into memory from the host GBA (which is running the host game—for instance, *Mario Kart Super Circuit*). See Figure 4.17 for a picture of an MBV2 device. Games with multiboot capability allow up to four players to participate in a game where only one of the players is using the actual game cartridge.

Figure 4.17

The Multi Boot Version 2 is a cable that connects a GBA to a PC using a parallel port adapter.

The great thing about the MBV2 is that it is a low-cost development device that complements Visual HAM wonderfully. HAM even includes built-in support for MBV2, as the transfer software is installed with HAM. In order to use the MBV2, you need to plug it into the parallel (printer) port on your PC, and then connect the blue cable to the link port on your GBA. Remove any cartridge from the GBA, so the cartridge slot is empty. Then turn on the GBA. Now, from within Visual HAM, you can choose to compile and run the program directly on the GBA.

Take a look at Figure 4.18, which shows the Project menu in Visual HAM, with the MBV2 menu item highlighted. Just be sure to compile the program first by pressing F5, and then you can send the program to the MBV2 device by pressing F8 (which causes the program to run on your GBA, so be sure to have your GBA power turned on before starting an MBV2 session).
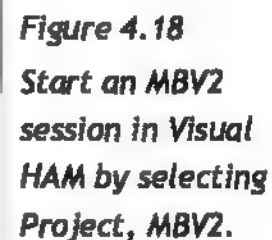
Figure 4.18
Start an MBV2 session in Visual HAM by selecting Project, MBV2.

Figure 4.19 shows the complete MBV2 retail kit, which comes with a disk containing the MBV2 driver and transfer software. Although aftermarket development accessories like the MBV2 are not available through retail channels, you may order the MBV2 kit on the Web, from sources such as http://www.lik-sang.com. You may also find other sources for this kit by going to a search engine and typing in "game boy advance multi-boot", which should return a list of sites selling this device (and others like it).

Figure 4.19
The MBV2 kit, showing the included software disk and retail packaging.

The Flash Advance Linker

The Flash Advance Linker (shown in Figure 4.20) is a parallel port device capable of reading and writing GBA cartridges. This device is used to copy your compiled GBA programs to a blank flash cartridge that is compatible with the GBA game cartridge slot. After writing the binary ROM for your program to the cartridge, you then remove it and insert it into your GBA, at which point it functions like any regular game cartridge.
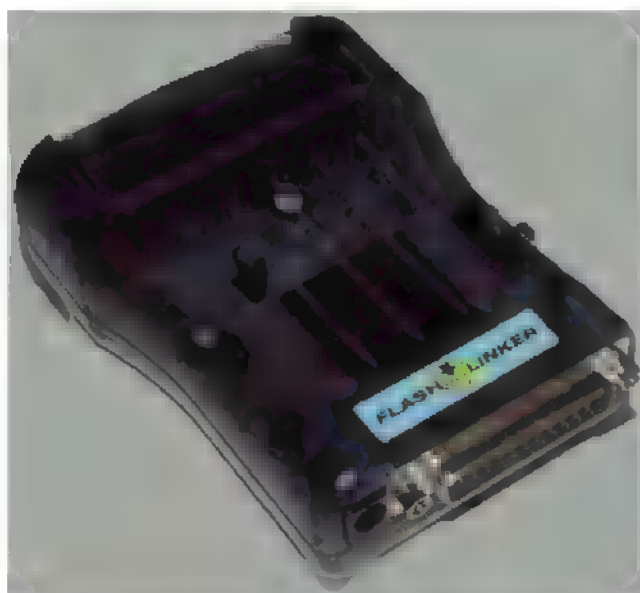
Figure 4.20
The Flash Advance Linker connects to a PC using a parallel port cable.

The flash cartridges used with the Flash Advance Linker come in varying sizes, including 64, 128, 256, 512, and even 1,024 Mbits. The standard 64M cartridge, shown in Figure 4.21, is one of the options available when you purchase your own Flash Advance Linker.

Figure 4.21

A rewritable flash cartridge comes with the Flash Advance Linker and is compatible with the GBA cartridge slot.

Unlike the MBV2, which connects directly to the parallel port and provides a link cable to your GBA, the Flash Advance Linker is a bulkier device, requiring a parallel cable to connect to your PC, and there is no direct connection to a GBA. Figure 4.22 shows the device with a parallel cable connected to it.

Figure 4.22

The Flash Advance Linker device with a flash cartridge inserted and a parallel cable attached.

Summary

Although the material in this chapter has been introductory in nature, providing the first working code samples for the book, you now have all the tools needed to write a rudimentary game by drawing simple shapes on the screen and detecting button input. In addition to providing a first glimpse into GBA programming, this chapter also provided an overview of the hardware accessories that allow you to develop and run programs directly on ▪ GBA.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter. The solution to each challenge is provided on the CD-ROM inside the folder for this chapter.

Challenge 1: Modify the Greeting program so that it displays a different message on the screen depending on which button has been pressed.

Challenge 2: Modify the DrawPixel program so that it moves the pixel around the screen and causes the pixel to bounce off the edges of the screen.

Challenge 3: Modify the FillScreen program by moving the for loops into a function called FillScreen that fills the screen with a specified RGB color.

Chapter Quiz

The key to the quiz may be found in Appendix D.

1. What language is featured in this book for writing Game Boy Advance programs?
    A. C++
    B. Basic
    C. C
    D. Prolog

2. What is the Hamlib function for displaying text on the screen?
    A. ham_DisplayText
    B. ham_DrawText

C. ham_PrintText

D. ham_SetText

3. What is the name of the GBA emulator used in this book (and distributed with HAM)?

    A. VisualBoyAdvance

    B. Visual Game Emulator

    C. GBA-EMU

    D. WinGBA

4. True or False: The HAM distribution comes with flash linker and multiboot software.

    A. True

    B. False

5. What is the display resolution of video mode 3?

    A. 320 x 240

    B. 260 x 180

    C. 120 x 80

    D. 240 x 160

6. What is the name of the software development kit *distribution* used in this book?

    A. Visual HAM

    B. HAM

    C. Hamlib

    D. DevKit-Advance

7. What is the color depth of the screen in video mode 3?

    A. 8 bits

    B. 16 bits

    C. 24 bits

    D. 32 bits

8. What is the name of the memory register used to change the video mode?

    A. REG_CHGMOD

    B. REG_MODECH

    C. REG_DISPCNT

    D. REG_DMA01

9. What is the name of the Hamlib function that initializes the text display system?
   A. ham_StartText
   B. ham_LoadText
   C. ham_BeginText
   D. ham_InitText

10. What parallel port device connects to the Ext port on the GBA in order to run programs directly on the GBA?
    A. Multi Boot Version 2 (MBV2)
    B. VisualBoyAdvance (VBA)
    C. Flash Advance Linker (FLA)
    D. Major League Baseball (MLB)

Part II

Being One
With The Pixel

Welcome to Part II of *Programming The Nintendo Game Boy Advance: The Unofficial Guide*. Part II includes three chapters that are dedicated entirely on programming the graphics system of the Game Boy Advance. These chapters cover the bitmap-based video modes, tile-based video modes (including coverage of backgrounds), and a chapter on sprites. Going into the first chapter of this part, you will learn to draw pixels on the screen in each video mode. Going out of the last chapter in this part, you will have learned how to scroll backgrounds and rotate, scale, and draw alpha-blended and animated sprites over backgrounds.

Chapter 5 — Bitmap-Based Video Modes

Chapter 6 — Tile-Based Video Modes

Chapter 7 — Rounding Up Sprites

Chapter 5

Bitmap-Based Video Modes

This chapter is an introduction to basic graphics programming for Game Boy Advance, explaining the various graphics modes that are available (with the pros and cons of each) and showing how to draw pixels, lines, circles, bitmaps, and other shapes on the screen in each mode. The GBA has six different video modes that you may use, each with different resolutions, color depths, and capabilities. This chapter will teach you how to use the bitmap-based video modes 3, 4, and 5. The tile-based modes (0, 1, and 2) are covered in the next chapter. One of the goals of this chapter is to provide you with a collection of functions for rendering graphics in any of the three bitmapped video modes, which you may then copy and paste into other projects.

Here are the main topics of this chapter:

- Introduction to bitmapped graphics
- Working with mode 3
- Working with mode 4
- Working with mode 5
- Printing text on the screen

Introduction to Bitmapped Graphics

Understanding bitmapped graphics is the key to GBA game development, because when it comes down to it, all games are based on bitmap images of one format or another. The most important thing to consider when choosing a bitmap format is the amount of loss. A lossy format has a high compression ratio that keeps the file small, but at the cost of image quality. Examples include JPG and GIF. When it comes to games, quality is absolutely essential, so a nonlossy format must be used. Examples include BMP, PNG, PCX, and TGA. You may have a preference for one or another nonlossy format, but you will need a tool to convert an image file into a GBA image.

Now, by *image* what I am really referring to is a raw format that is actually compiled into your program. This differs greatly from what you may be used to, where in a standard operating system (Windows, Linux, Mac) you may store game graphics in one or more files and load them when needed. However, the GBA doesn't have a file system—everything must be stored inside the ROM! When I was first learning about GBA graphics programming, I thought perhaps the GBA used a proprietary bitmap file format stored in the ROM in a special resource of some kind. But that is not right; it's actually much simpler than that. The bitmaps used for backgrounds, tiles, and sprites in a GBA program must be converted to a C file as an array of numbers! Archaic, isn't it? Well, that's the console world for you. What this means, unfortunately, is that anytime you need to make a change to game graphics, you must run a utility program to convert the bitmap file to a source code file, such as monster.c.

> *In addition to a C byte array, graphics may be converted to raw binary format and linked directly into a game. For the sake of clarity, this book uses C arrays exclusively. In a practical sense, there is no real advantage one way or the other, performance-wise, although it is much easier to just include the C array for a bitmap in a project.*

For example, here is the file from the ShowPicture program presented in Chapter 3:

```
//
// bg (38400 uncompressed bytes)
//
extern const unsigned char bg_Bitmap[38400] = {
0x4f, 0x3b, 0x23, 0x23, 0x2c, 0x1f, 0x22, 0x2d, 0x2f, 0x2f, 0x2f, 0x2f,
```

```
0x31, 0x2f, 0x2d, 0x2f, 0x23, 0x1d, 0x2d, 0x4c, 0x37, 0x38, 0x2f, 0x2d,
0x22, 0x1f, 0x2f, 0x2d, 0x2d, 0x1d, 0x22, 0x2d, 0x23, 0x23, 0x33, 0x2f,
0x2f, 0x2d, 0x34, 0x2f, 0x2f, 0x2d, 0x2c, 0x23, 0x23, 0x2d, 0x2d, 0x2d,
0x22, 0x1f, 0x2f, 0x2b, 0x23, 0x2d, 0x2e, 0x2d, 0x1d, 0x23, 0x38, 0x2b,
.
.
.
0xe1, 0xe4, 0xc7, 0xbf, 0xc3, 0xf2, 0xf2, 0xf6, 0xf6, 0xf7, 0xf8, 0xf9,
0xf8, 0xf8, 0xf8, 0xf8, 0xf9, 0xf9, 0xf8, 0xf8, 0xf8, 0xf8, 0xf2, 0xf8,
0xf8, 0xf8, 0xf2, 0xf6, 0xf6, 0xf2, 0xf2, 0xf2, 0xf2, 0xf6, 0xf2, 0xd9,
0xe1, 0xd6, 0xd9, 0xd9, 0xd9, 0xd9, 0xc9, 0xf6, 0xf6, 0xf2, 0xf6, 0xf8,
0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf9, 0xf9, 0xf9
};
```

I have listed only a portion of the actual file, which is quite large. This bitmap image was originally called bg.bmp (shown in Figure 5.1) and was converted by a program called gfx2gba.exe. This program specifically converts Windows .bmp files to GBA format. Actually, I shouldn't call it GBA format, because any C program could use it.

Figure 5.1
The bg.bmp bitmap image.

As you can see from the header, there are 38,400 bytes in this bitmap file. Regardless of the source image format, this is a raw format, just an array of 16-bit hexadecimal numbers, each of which represents a single pixel. If you do a little deductive mathematics, you can make an educated guess on the image size. A screen resolution of 240 x 160 = 38,400 pixels, so this must be a full-screen background image used in a mode 3 program. Of course, bitmaps are not limited to the screen resolution. Indeed, a bitmap can be quite huge, particularly when working with an animated sprite. For example, Figure 5.2 shows an animated explosion sprite, with a resolution of 524 x 142, which is much larger than the physical screen. However, when displayed on the screen, the explosion is 64 x 64. That's actually a rather large sprite for the GBA screen (approximately one-fourth of the screen),

but it looks terrific when animated! I'll cover sprites in more detail in Chapter 7, "Rounding Up Sprites."
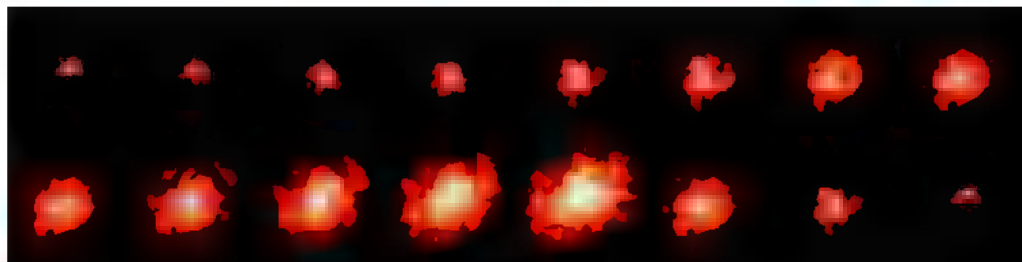
Figure 5.2

An animated sprite comprises multiple frames stored in a single bitmap file.

Selecting the Ideal Video Mode

When it comes to designing a new game, the resolution for your game is an important consideration early on, because the modes supported on the GBA differ greatly. Table 5.1 shows the three bitmap modes and their settings.

Table 5.1   Video Modes

Mode	Resolution	Color Depth	Double Buffered
3	240 x 160	15	No
4	240 x 160	8	Yes
5	160 x 128	15	Yes

Hardware Double Buffer

Modes 4 and 5 require less memory for the video buffer, so they are able to use two buffers for high-speed flicker-free screen updates. Mode 3, on the other hand, is both high in resolution and high in color depth, so it has enough video memory for just a single video buffer; thus double buffering is not available. However, there's no reason why you can't create your own double buffer in EWRAM and then perform a fast copy to the video buffer during the vertical retrace. A double buffer directly in video memory is the fastest method, of course, but using EWRAM is fast enough for most needs. If IWRAM were larger than 32 KB, it would be a great place to store the double buffer, but we need 75 KB for a mode 3 buffer. Since EWRAM has 256 KB available, it is the only viable option. Even if you are skilled enough to divide the second buffer between the 32 KB IWRAM and the remaining 20 KB or so

leftover in video memory, that still isn't enough memory for the mode 3 buffer. I'll show you how to create ▪ mode 3 double buffer later in this chapter.

Horizontal and Vertical Retrace

The screen refreshes at a fixed interval in the horizontal and vertical directions. What this means is that there is a short period after the LCD draws a horizontal line while it repositions to the next line and a somewhat longer (but still relatively short) blank period after the last pixel has been drawn while the video chip repositions back up to the upper-left corner of the screen. When dealing with a cathode ray tube (CRT) monitor for a PC, or perhaps ▪ television, there are physical electron guns shooting electrons through the phosphorous layer of the screen, causing each pixel to light up for a brief period. Some of the older displays and TVs had a problem with ghosting because the phosphorous layer's pixels would remain lit too long, but modern displays don't usually have this problem any longer. The vertical retrace is really the only thing we're interested in for game programming, because this provides a window of opportunity. Any screen writes done during the vertical refresh are displayed as a whole, providing a nice consistent screen (and even frame rate).

Working with Mode 3

Video mode 3 is probably the most appealing mode, because it is the highest quality mode available on the GBA, with a resolution of 240 x 160 and a 15-bit color depth providing up to 32,767 colors on the screen at once. What this means is that your game's graphics will look extraordinary, with beautiful shades and intricate backgrounds. In contrast, mode 4 is limited to 256 colors, and while mode 5 also has 15-bit color, it is limited in resolution. The only problem with mode 3 is that you must provide your own double buffer. This isn't ▪ problem, but the buffer does take away from available EWRAM that may be needed in the game (for instance, sound effects and music). However, let's just try a few things and see how it goes. This may be a factor in your decision to go with one video mode over another. For instance, if you are writing a high-speed 3D game that needs lots of memory, I would recommend mode 5 because it is much, much faster, due to the lower resolution, while still maintaining quality.

Drawing Basics

Let's dive right into the code for mode 3 and get something up on the screen. You have already seen an example of mode 3 in action from the DrawPixel and FillScreen programs in Chapter 4. However, one of those used Hamlib and the other dealt only with pixels. Now it's time for your formal training in pixel management.

Drawing Pixels

It's very easy to draw pixels in mode 3, because each pixel is represented by a single element of the video buffer (which comprises unsigned shorts). To draw a pixel, use an algorithm that has been passed down from one game programmer to another throughout history:

```
videoBuffer[y * SCREENWIDTH + x] = color;
```

Here's a complete reusable function for drawing pixels in mode 3:

```
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = c;
}
```

The Mode3Pixels program demonstrates how to draw random pixels on the mode 3 screen. Create a new project in Visual HAM called Mode3Pixels and replace the default code in main.c with the following code listing:

```
////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Pixels Project
// main.c source code file
////////////////////////////////////////////////////////


//add support for the rand function
#include <stdlib.h>


//declare the function prototype
```

```c
void DrawPixel3(int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
        x = rand() % 240;
        y = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);
```

```
        DrawPixel3(x, y, color);

    }


    return 0;

}


/////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}
```

The output from the Mode3Pixels program is shown in Figure 5.3.
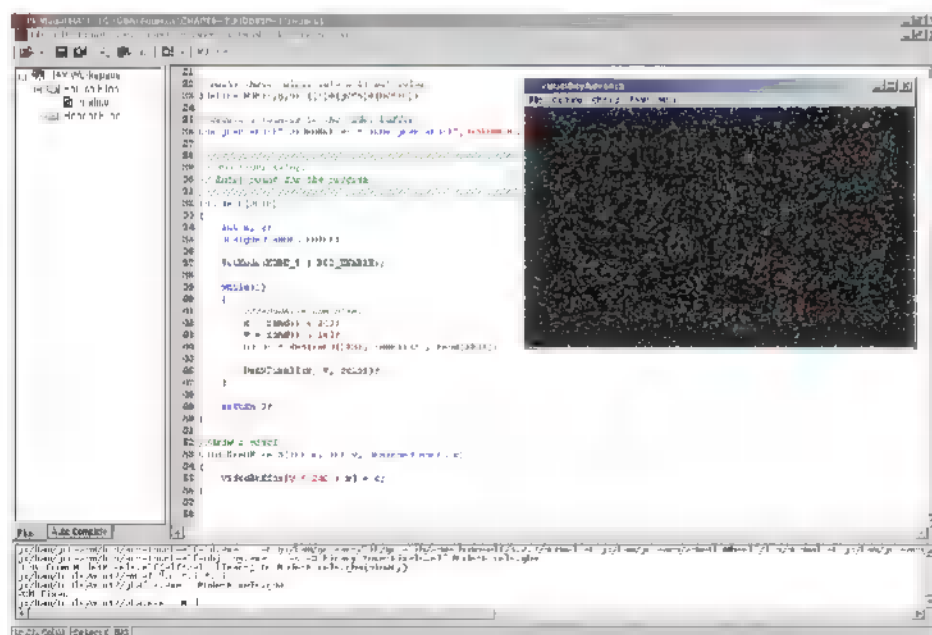
Figure 5.3
The Mode3Pixels
program draws
random pixels on
the GBA screen.

Drawing Lines

One of the most commonly requested functions for any computing platform is an algorithm for drawing lines. Most of us would prefer to have an SDK that has hardware support for line drawing. It would be great if the GBA had such a routine built into the hardware, because that would be extremely fast and would allow us to write high-speed polygon-type games

(which are based on triangles, which are created with a high-speed line-drawing function). There are several line-drawing algorithms out there, but by far the most common is Bresenham's line drawing algorithm. Since this isn't a book about computer science theory per se, I'm not going to provide a detailed overview of the theory behind this algorithm. It is enough to just use it and assume that it works—by watching the results. The Mode3Lines program demonstrates how to use the DrawPixel3 function to do something more useful than pixel plotting. This project is located on the CD-ROM under \Sources\Chapter05\Mode3Lines.

```c
///////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Lines Project
// main.c source code file
///////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//add support for the rand function
#include <stdlib.h>


//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawLine3(int, int, int, int, unsigned short);


//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


//packs three values into a 15-bit color
```

```c
#define RGB(r,g,b)  ((r)+(g<<5)+(b<<10))


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


/////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////
int main(void)
{
    int x1,y1,x2,y2;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

    while(1)
    {
        x1 = rand() % 240;
        y1 = rand() % 160;
        x2 = rand() % 240;
        y2 = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawLine3(x1,y1,x2,y2,color);
    }

    return 0;
}


/////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////////////
```

```c
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}


/////////////////////////////////////////////////////////////
// Function: DrawLine3
// Bresenham's infamous line algorithm
/////////////////////////////////////////////////////////////
void DrawLine3(int x1, int y1, int x2, int y2, unsigned short color)
{
    int i, deltax, deltay, numpixels;
    int d, dinc1, dinc2;
    int x, xinc1, xinc2;
    int y, yinc1, yinc2;

    //calculate deltaX and deltaY
    deltax = abs(x2 - x1);
    deltay = abs(y2 - y1);

    //initialize
    if(deltax >= deltay)
    {
        //If x is independent variable
        numpixels = deltax + 1;
        d = (2 * deltay) - deltax;
        dinc1 = deltay << 1;
        dinc2 = (deltay - deltax) << 1;
        xinc1 = 1;
        xinc2 = 1;
        yinc1 = 0;
        yinc2 = 1;
    }
    else
```

```
{
    //if y is independent variable
    numpixels = deltay + 1;
    d = (2 * deltax) - deltay;
    dinc1 = deltax << 1;
    dinc2 = (deltax - deltay) << 1;
    xinc1 = 0;
    xinc2 = 1;
    yinc1 = 1;
    yinc2 = 1;
}


//move the right direction
if(x1 > x2)
{
    xinc1 = -xinc1;
    xinc2 = -xinc2;
}
if(y1 > y2)
{
    yinc1 = -yinc1;
    yinc2 = -yinc2;
}


x = x1;
y = y1;


//draw the pixels
for(i = 1; i < numpixels; i++)
{
    DrawPixel3(x, y, color);

    if(d < 0)
    {
```

```
            d = d + dinc1;

            x = x + xinc1;

            y = y + yinc1;

        }
    else
    {

            d = d + dinc2;

            x = x + xinc2;

            y = y + yinc2;

        }
    }
}
```

The output from the Mode3Lines program is shown in Figure 5.4.
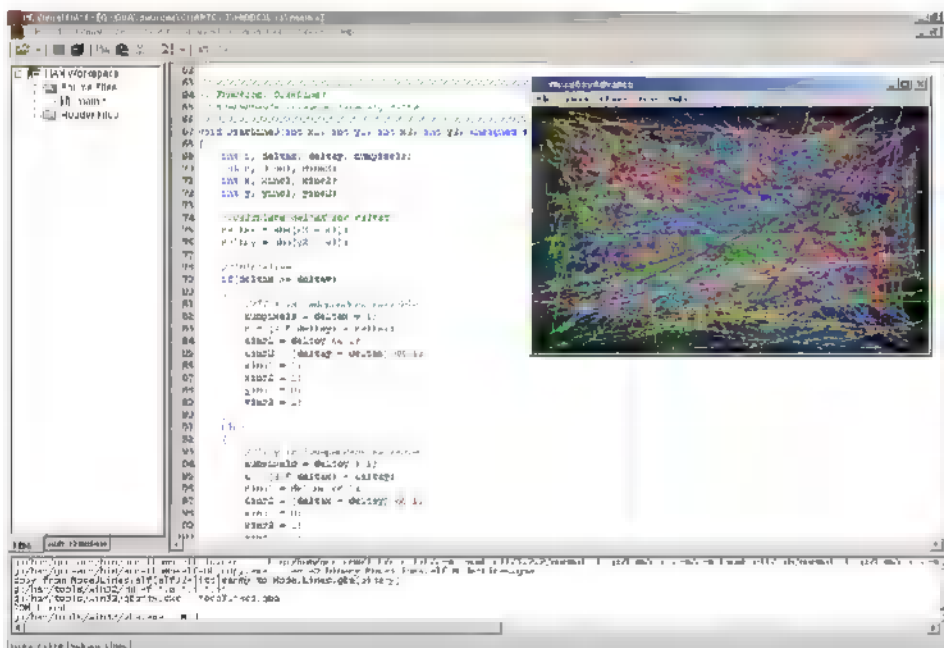
Figure 5.4
The Mode3Lines
program draws
random lines on
the GBA screen.

Making GBA Programs Multi-Bootable

At the top of the code listing for the Mode3Lines program is a rather strange-looking couple of lines:

```
#define MULTIBOOT int __gba_multiboot;
MULTIBOOT
```

These two lines of code are very important, so I will mention them in each chapter from this point forward (since many readers prefer to jump to their favorite chapters, and may not necessarily read through the whole book). Multi-boot is a feature of the GBA that allows it to run multi-player games using link cables, where a single GBA has a multi-player game cartridge, while the other players may go without. For example, *The Legend of Zelda: A Link To The Past* includes a four-player mini-game called *Four Swords* that can be played with two, three, or four players.

To run a multi-boot game, simply plug in the link cables, remove any cartridges from the client players, and leave in the game cartridge for the host player. When the GBA detects that there is no cartirdge, it goes into multi-player mode, and attempts to download a ROM over the link cable. The Multi-Boot Version 2 (MBV2) cable, available from http://www.lik-sang.com, takes advantage of this feature by allowing you to test your GBA programs using a special utility program that sends a ROM through the PC's parallel port to the GBA, in the same manner that a multi-player game ROM is transferred from a host GBA to client players.

The definition

```
#define MULTIBOOT int __gba_multiboot;
```

is simply a compiler directive that creates an int variable called __gba_multiboot (note the double underscores in front, and single underscore between gba and multiboot). The HAM SDK recognizes this variable declaration as an instruction to make the ROM image multi-bootable, which adds support for MBV2.

However, just creating the definition using #define doesn't make a game bootable. Rather, you must use the definition, which is what the second line does:

```
MULTIBOOT
```

If you prefer, you may simply insert a single line at the top of the program like this:

```
int __gba_multiboot;
```

and that will suffice. However, everyone in the GBA community is using MULTIBOOT, and it is ■ standard definition in the popular header files (such as gba.h and mygba.h, included with many public domain GBA games).

The only condition on using MULTIBOOT is that it must be included at the top of the source code listing, even before the include statements, and the ROM may not be larger than 256 KB. Unfortunately, that is the limit of the RAM on the GBA. Remember, there is no cartridge in the GBA, with multiple megabytes of space available, just the RAM! If you are working on ■ sizeable game, the MBV2 may not work for a project if the ROM is too big.

However, the sample programs in this book are all fairly small, so each sample program from this point forward will include the MULTIBOOT statement. If you would like to test a multi-boot program yourself, you will first need the MBV2 device. It is around $30.00, so if you plan to do a lot of GBA development, I highly recommend buying one from Lik-Sang or another distributor. If you do have a MBV2, and want to try it out immediately, I have included a folder on the CD-ROM called \MultiBoot. This folder includes ■ compiled version of every sample program in the book, ready to be run via MBV2. The MB.EXE is also in the folder, along with a convenient batch file called multiboot.bat that uses the appropriate options.

Note that an extra step is required for Windows 2000 or XP. First, you must install the userport driver, due to the way the MBV2 adapter works. I have included the userport driver in the \MultiBoot folder under \MultiBoot\Win_2K_XP. First, copy the userport.sys file to your \WINNT\SYSTEM32\DRIVERS folder. Next, run userport.exe to configure the parallel port as appropriate for your PC. Normal configurations will use LPT1 at address 378-37F. You may read the readme.txt file or the userport.pdf file (using Adobe Acrobat Reader) for more help with the userport driver. Once you have run the userport.exe file and configured the port address, you may then run the MB.EXE program to transfer a program to the GBA and watch it run on the actual hardware. Depending on your system, you may need to run the userport.exe program each time you start using the MBV2 (on a per-boot basis, or each time you open the Command Prompt, depending on the version of Windows you are using— and note that userport is not needed for Windows 98/ME).

To run ■ program on your GBA, first remove any cartridge, then turn on your GBA. The Game Boy logo will appear, and then it will wait for a signal from MBV2. Now, open a Command Prompt (Start, Run, cmd.exe). CD to the \MultiBoot folder on the CD-ROM (or copy the folder to your hard drive first, and CD to that folder), then type

```
multiboot Mode3Lines
```

without the extension, because .gba is added to the filename by the batch file. For reference, the multiboot.bat file includes the following command:

```
mb -w 50 -s %1.gba
```

Note that I have tested all of the ROMs in the MultiBoot folder on an actual GBA, using the MBV2 cable, and verified that they all work, at least with the configuration on my development PC (which I have been describing here). Some of the sample programs look amazing running on the GBA! For instance, the sample programs from Chapter 7, "Rounding Up Sprites." While I wouldn't encourage you to skip ahead so soon, the sprite demos are particularly impressive, such as TransSprite, which displays five bouncing balls that alternate between solid and translucent, using alpha blending! However, the sample programs from this chapter and all of them, really, are loads of fun to demo on an actual GBA. If you don't own a MBV2, I apologize for in After running the command, you should see a message that looks something like this:

```
Parallel MBV2 cable found: hardware=v2.0, firmware=v4
EPP support found.
Looking for multiboot-ready GBA...
Sending data. Please wait...
CRC Ok - Done.
```

If the GBA is not turned on, or if the parallel port has not been configured properly, you may get an error message such as the following.

```
ERROR - No hardware found on selected parallel port!
Check parallel port number, make sure cable is connected to port and to GBA,
▮ make sure GBA is turned on.
```

I recommend setting it to an Enhanced Parallel Port (EPP) in your PC's BIOS setup. If your PC doesn't have a parallel port with EPP mode, that's okay, the MBV2 will still work fine on it, but you may have to adjust the wait period. I have used 50 milliseconds, because that is the value that worked best for my PC. If you keep getting timeout errors, you might bump the wait to -w 100, although if that still doesn't work, I would suspect ▮ problem with your ports or the userport driver.

Running the sample programs through the MBV2 directly on your GBA is a fascinating experience. Watching your own games run on the actual handheld console is nothing short of thrilling, and you will amaze your friends and relatives by showing them your game running on an actual GBA! That is why I highly recommend buying a MBV2 adapter. They are $30.00, and only available via mail order. Shipping is rather high from Asia, so you may want

to save up and pick up a few things to combine shipping (for instance, order a Flash Advance Linker at the same time). Even with a Flash Advance Linker, I highly recommend ■ MBV2, simply because it's fast and easy to use. You can compile and run programs via MBV2 directly from within Visual HAM (see the Projects menu). Sometimes it can be a little frustrating to get the MBV2 to work flawlessly every single time. If you are having trouble getting it to work, I recommend you check the MBV2 FAQ at http://www.devrs.com/gba/files/mbv2faqs.php. Of course, you may also browse for "MBV2" on Google.com or another search engine to find additional resources.

Drawing Circles

Bresenham was a genius, because he not only provided us with a cool line-drawing algorithm, but he also created a great circle-drawing algorithm too! As with the line algorithm, if you want the theory behind it, I would recommend picking up ■ book on graphics theory or searching the Web for a suitable tutorial. The Mode3Circles program demonstrates how to use the Bresenham algorithm for drawing circles in mode 3.

```c
////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Circles Project
// main.c source code file
////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//add support for the rand function
#include <stdlib.h>


//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawCircle3(int, int, int, int);


//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
```

```c
#define MODE_3 0x3

#define BG2_ENABLE 0x400


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


//packs three values into a 15-bit color
#define RGB(r,g,b)  ((r)+(g<<5)+(b<<10))


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////////////////////
int main(void)
{
    int x, y, r;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

    while(1)
    {
        x = rand() % 240;
        y = rand() % 160;
        r = rand() % 50 + 10;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawCircle3(x, y, r, color);
    }

    return 0;
```

```
}

////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}


////////////////////////////////////////////////////////
// Function: DrawCircle3
// Bresenham's infamous circle algorithm
////////////////////////////////////////////////////////
void DrawCircle3(int xCenter, int yCenter, int radius, int color)
{
    int x = 0;
    int y = radius;
    int p = 3 - 2 * radius;
    while (x <= y)
    {
        DrawPixel3(xCenter + x, yCenter + y, color);
        DrawPixel3(xCenter - x, yCenter + y, color);
        DrawPixel3(xCenter + x, yCenter - y, color);
        DrawPixel3(xCenter - x, yCenter - y, color);
        DrawPixel3(xCenter + y, yCenter + x, color);
        DrawPixel3(xCenter - y, yCenter + x, color);
        DrawPixel3(xCenter + y, yCenter - x, color);
        DrawPixel3(xCenter - y, yCenter - x, color);

        if (p < 0)
            p += 4 * x++ + 6;
        else
```

```
        p += 4 * (x++ - y--) + 10;

    }

}
```

The output from the Mode3Circles program is shown in Figure 5.5.
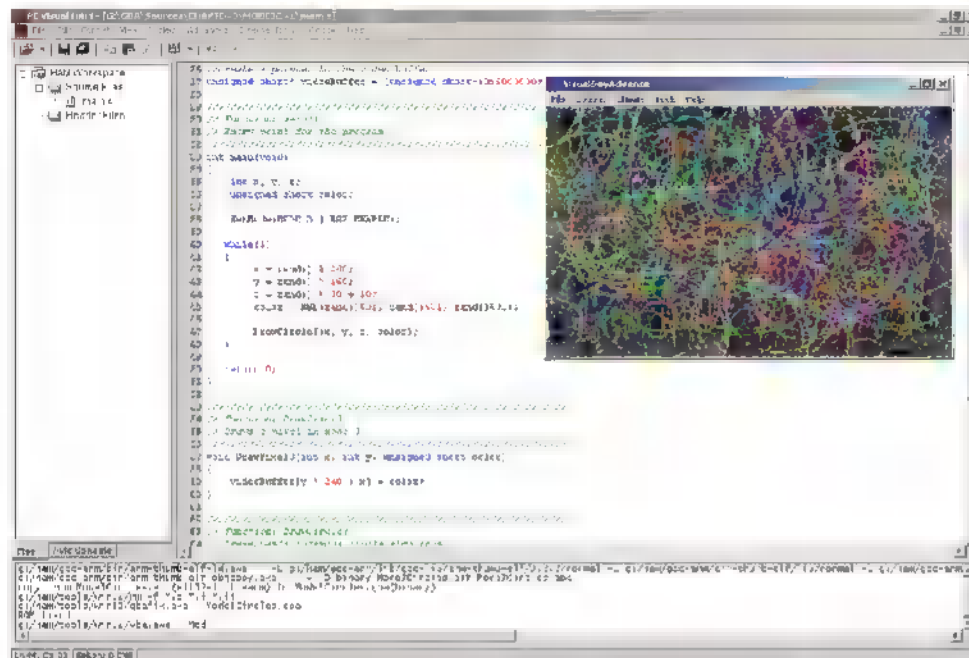
Figure 5.5

The Mode3Circles program draws random circles on the GBA screen.

Drawing Filled Boxes

How about something a little more interesting? It's truly amazing what you can do after you have the basic pixel-plotting code available! The Mode3Boxes program draws filled boxes on the screen. While I could have used Bresenham's line algorithm, that would have been grossly wasteful, because that algorithm is only useful for diagonal lines. When it comes to straight lines in a filled box, all you need to do is throw in a couple of loops and draw the pixels. This project is called Mode3Boxes and is located on the CD-ROM under \Sources\Chapter05\Mode3Boxes.

```
///////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Boxes Project
// main.c source code file
///////////////////////////////////////////////////////
```

```c
#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//add support for the rand function
#include <stdlib.h>


//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawBox3(int, int, int, int, unsigned short);


//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x1, y1, x2, y2;
    unsigned short color;


    SetMode(MODE_3 | BG2_ENABLE);
```

```
        while(1)
        {
                x1 = rand() % 240;
                y1 = rand() % 160;
                x2 = x1 + rand() % 60;
                y2 = y1 + rand() % 60;
                color = RGB(rand()%31, rand()%31, rand()%31);


                DrawBox3(x1, y1, x2, y2, color);
        }


        return 0;
}


//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
        videoBuffer[y * 240 + x] = color;
}


//////////////////////////////////////////////////////////////
// Function: DrawBox3
// Draws a filled box
//////////////////////////////////////////////////////////////
void DrawBox3(int left, int top, int right, int bottom,
        unsigned short color)
{
        int x, y;


        for(y = top; y < bottom; y++)
```

```
        for(x = left; x < right; x++)


            DrawPixel3(x, y, color);


}
```

The output from the Mode3Boxes program is shown in Figure 5.6.
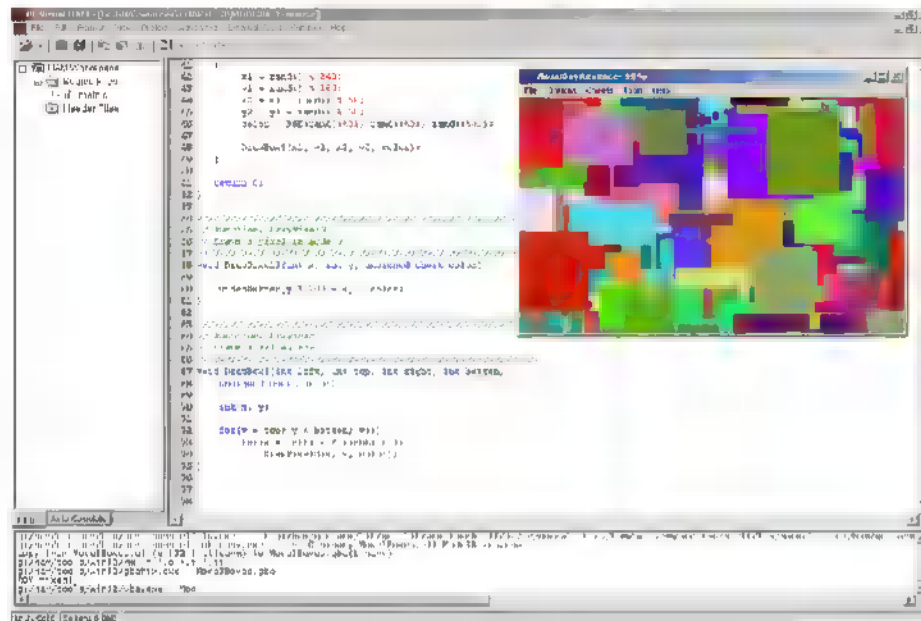
Figure 5.6

The Mode3Boxes program draws random filled boxes on the GBA screen.

Drawing Bitmaps

Now for something really interesting (as if the last few sections haven't already been?). When it comes down to it, you can make a game using just pixels, lines, and other vector objects. But when you want to do something really cool, you need bitmaps. I'm not covering sprites until later (Chapter 7), but for now I'd like to introduce you to bitmap images and show you how to draw them on the screen.

Converting 8-Bit Bitmaps to 15-Bit Images

The first thing you need to do is create the source bitmap to display and tweak the color depth so it is saved in 8-bit, 256-color mode. This does reduce the number of possible colors in 15-bit mode 3 and mode 5, but the gfx2gba program can only read 8-bit images (and since it is the most popular tool, we'll just work around the limitation). Note that the GBA only uses 15 out of 16 bits in a 555 format (that's Blue/Green/Red, or more commonly referred to as BGR), and the last bit is ignored.

There is a good bonus to using 8-bit images for all of your source artwork, because then you need to only keep track of ■ single collection of artwork and need not worry about keeping two versions (both 8-bit and 16-bit image files).

Converting Bitmap Images to Game Boy Format

The GBA doesn't have a file system of any kind, so you must embed the bitmaps (as well as sound effects, music, and any other data files) into source code as an array. The nice thing about this is that you really don't have to worry about writing code to load ■ bitmap, sound, or other resource, because it's immediately available to your program as ■ C-style array—such as the one I showed you at the beginning of this chapter. Several tools are available in the public domain to convert a bitmap file to a GBA source code file; the most common of these is gfx2gba.exe. There is no formal support or installer for this tool—it's just ■ public domain program (like most of the utility programs for the GBA). The gfx2gba.exe program is automatically installed with HAM, so there is no need to download it. The file is located in the \ham\tools\win32 folder, on whatever drive you installed HAM to.

A convenient batch file included with HAM called startham.bat sets up ■ path to this folder so you can run the utility programs from the command-line prompt. Just open a command window by clicking Start, Run and typing in "cmd". Then click on the OK button. A command window should open, providing a prompt that is familiar to those of us who once used MS-DOS. You will need to change to the folder where your bitmap file is located in order to convert it. Of course, you can also type in a fully qualified pathname to your source bitmap, but I prefer to run gfx2gba.exe while in the source folder already. You can move to the folder for the next sample project in this chapter by typing "CD \Sources\Chapter05\Mode3Bitmap". I am assuming here that you have copied the \Sources folder off the CD-ROM to the root of your hard drive. Of course, you will want to modify that pathname to match the folder you are using for book projects.

To convert ■ bitmap using gfx2gba.exe for mode 3 (which is not palettized), you will want to type in this command:

```
gfx2gba -fsrc -c32k filename.bmp
```

The source file could also be a .pcx image. The important thing is that the image must be saved in 8-bit, 256-color mode, because that is the only format that gfx2gba supports. Now for a short explanation of the options used. The -fsrc option specifies that the output should be C source code. The -c32k option tells gfx2gba to output nonpalettized 15-bit pixels (without this option, only 8-bit palettized colors are used, which won't work in mode

3 or mode 5). Finally, the last option specifies the source file to convert. Now that you know this, I can show you the command to convert the mode3.bmp file for the upcoming project:

```
gfx2gba -fsrc -c32k mode3.bmp
```

This command produces one output file called mode3.raw.c. The mode3.raw.c file looks similar to the file I showed you earlier:

```
//
// mode3 (76800 uncompressed bytes)
//
const unsigned short mode3_Bitmap[38400] = {
0x7717, 0x7293, 0x69cd, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f,
0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f,
0x6e0f, 0x6e0f, 0x7293, 0x7717, 0x6e51, 0x6e0f, 0x7717, 0x6e51,
0x656a, 0x69cd, 0x61aa, 0x58c3, 0x6e51, 0x6e51, 0x7293, 0x7717,
0x7bbc, 0x7b5a, 0x6a2f, 0x6a2f, 0x6e51, 0x69cd, 0x656a, 0x6a2f,
0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x7717, 0x7b9b,
0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7759, 0x7293,
0x6e51, 0x6a2e, 0x6a2f, 0x6a72, 0x7293, 0x6127, 0x58c3, 0x58c3,
 . . .
```

Note that all full-size 15-bit bitmaps will be 76,800 bytes in length, while the file I showed you earlier was a 38,400-byte image—which was a palettized image for mode 4. The palette file is something you have not seen yet, but I will cover that in the section on mode 4.

Drawing Converted Bitmaps

After you have converted a bitmap file to C source, you can then directly use the image in a GBA program without any real work on your part. That is one nice aspect of GBA graphics programming—all the work is done up front. Here's the source code for the Mode3Bitmap program:

```
////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Bitmap Project
```

```
// main.c source code file
//////////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//include files
#include <stdlib.h>
#include "mode3.raw.c"


//declare the function prototype
void DrawPixel3(int, int, unsigned short);


//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;


    SetMode(MODE_3 | BG2_ENABLE);
```

```
                //display the bitmap
        for(y = 0; y < 160; y++)
                for(x = 0; x < 240; x++)
                        DrawPixel3(x, y, mode3_Bitmap[y * 240 + x]);


        //endless loop
        while(1)
        {
        }


        return 0;
}


//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
        videoBuffer[y * 240 + x] = c;
}
```

The output from the Mode3Bitmap program is shown in Figure 5.7.

There's a faster way to draw bitmaps using DMA, due to a high-speed hardware-based memory copy feature. Since it's based in hardware, you can expect it to be several times faster than displaying the bitmap one pixel at a time. I will cover DMA features in Chapter 8, "Using Interrupts and Timers."
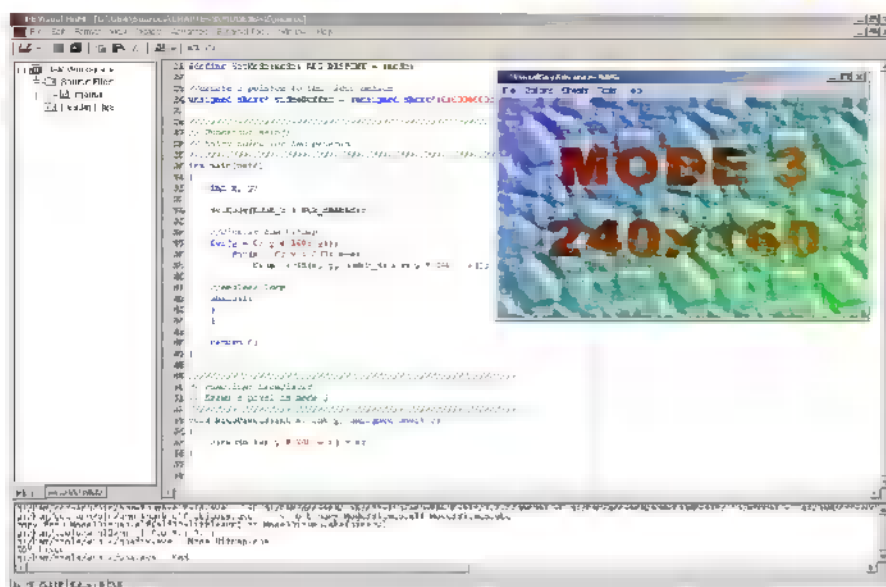
Figure 5.7
The Mode3Bitmap program draws a bitmap image on the GBA screen.

Working with Mode 4

Video mode 4 is an 8-bit (palettized) 240 x 160 display with a more difficult pixel format where two pixels are packed into a 16-bit value, and there is no other way to read or write values in this memory buffer except 16 bits at a time (which is an unsigned short value).

Dealing with Palettes

Mode 4 is an 8-bit palettized display mode, which means that there are only 256 total colors available, and each color is stored in a lookup table called the *palette*. Here is a sample mode 4 palette output by the gfx2gba program:

```
const unsigned short mode4_Palette[256] = {
0x0001, 0x0023, 0x0026, 0x0086, 0x002a, 0x008a, 0x00aa, 0x010a,
0x00a9, 0x00ad, 0x00ae, 0x014e, 0x3940, 0x190b, 0x44e0, 0x4522,
0x25c0, 0x1e20, 0x3181, 0x2dc1, 0x2a23, 0x15cb, 0x3604, 0x3629,
0x4183, 0x45a5, 0x41e5, 0x4206, 0x3e27, 0x4627, 0x4228, 0x420c,
0x5481, 0x58c3, 0x5525, 0x6127, 0x4d86, 0x4dc7, 0x5186, 0x5988,
.
.
.
0x5fb5, 0x6357, 0x6396, 0x63b7, 0x639a, 0x6b36, 0x6b57, 0x6f37,
0x7357, 0x6f58, 0x7359, 0x7759, 0x7b5a, 0x6b78, 0x6f99, 0x6b99,
0x6fba, 0x739a, 0x7779, 0x7b9a, 0x6fdb, 0x73dc, 0x77bb, 0x7b9b,
```

```
0x7bbc, 0x7bdc, 0x7bdd, 0x77fd, 0x7fde, 0x7bfe, 0x7ffe, 0x77bf,
0x7bdf, 0x7fff, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
};
```

The palette file isn't nearly as long as the raw file, but I have truncated it to save space. You may open the files yourself using Notepad or Visual HAM to see the full listing. When working with mode 4, you need to set up the palette table first before drawing anything, because most likely the palette entries are all 0.

The palette memory on the GBA is a 256-byte section of memory located at memory address 0 x 5000000. To use the palette, then, you'll need to create a pointer to this location in memory:

```
unsigned short* paletteMem = (unsigned short*)0x5000000;
```

Setting the palette entries is then simply a matter of setting the value for each element of the 256 paletteMem array. For instance, this code sets all the colors of the palette to white:

```
for (n = 0; n < 256; n++)
    paletteMem[n] = RGB(31, 31, 31);
```

Most games that use mode 4 have a master palette that is used by all the graphics in the game. The reason this is necessary is because you can only have one palette active at ■ time. Now, you could easily change the palette for each game level, and that is what most games do! There's no need to limit your creativity just because there are only 256 colors available at a time. By having several (or a dozen) palette arrays available, one per level of the game, your game can be very colorful and each level can be distinctive. Note that it's common practice to keep palette entry ▯ set to black (that is, 0 x 00) because that is often used as the color for transparency. Remember, even the sprites in your game will use this palette (unless the sprites each have their own 16-color palette—more on that in Chapter 7, "Rounding Up Sprites"). Figure 5.8 shows an illustration that helps to explain the difference between mode 3 (15-bit) and mode 4 (8-bit).

8 BIT PIXEL

Color Palette Table

0 1 2 3 · · · 255

References
15-Bit Color →

15 BIT PIXEL

BGR

Pixel = 0x01

Pixel = 0x0FEE
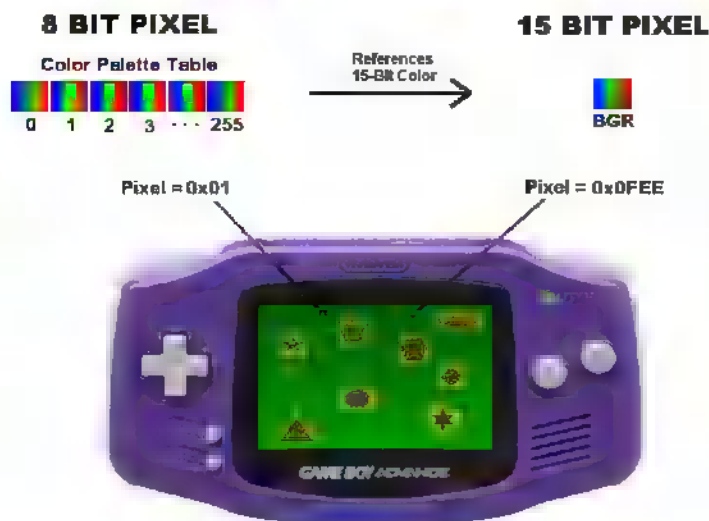
Figure 5.8

*Comparison of 8-bit and
15-bit display modes.*

Drawing Pixels

As I mentioned briefly a moment ago, mode 4 uses a more difficult pixel-packing method than the straightforward method used in modes 3 and 5 (where each pixel is represented by a single unsigned short). The reason why mode 4 is more difficult is because it's an 8-bit mode, and the video display system on the GBA can only handle 16-bit values. The video chip literally blasts two pixels at a time on the LCD screen, which is great for speed; however, the drawback is that setting pixels is something of a bottleneck. Basically, each even pixel is stored in the lower half of the unsigned short value, while each odd pixel is stored in the upper half. Figure 5.9 illustrates the point.

16-Bit Unsigned Short (15-Bit Pixel)

Figure 5.9

*Mode 4 pixels are
packed in twos for
every 16-bit number
in the video buffer.*

Bits 0-7	Bits 8-15
Even Pixels **(0, 2, 4, 6, 8 ...)**	**Odd Pixels** **(1, 3, 5, 7, 9 ...)**

In order to set a pixel, one must first read the existing 16-bit value, combine it with the new pixel value, and then write it back to the video buffer. As you can deduce, three steps are essentially required to set a pixel instead of just one step—which is a given when

working with transparent sprites. Fortunately, the hardware sprite handler takes care of that, but this support doesn't necessitate ignoring how this video mode works for your own needs, so I'm going to show you how to write a pixel in this mode.

First, read the existing unsigned short value, dividing the x value by 2:

```
unsigned short offset = (y * 240 + x) >> 1;
pixel = videoBuffer[offset];
```

Next, determine whether x is even or odd and AND'ing x with 1:

```
if (x & 1)
```

Finally, if x is even, then copy the pixel to the lower portion of the unsigned short. In order to do this, you must shift the color bits left by 8 bits so they can be combined with a number that is right-aligned, like this:

```
videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
```

If x is odd, then copy it to the upper portion of the number, without worrying about bit shifting, like so:

```
videoBuffer[offset] = (pixel & 0xFF00) + color;
```

This is obviously slower than simply writing a pixel to the video buffer. However, mode 4 has an advantage of being fast when using hardware-accelerated blitting functions and DMA, because twice as many pixels can be copied to video memory. What mode 4 is doing is essentially packing two pixels into the same space occupied by one pixel in modes 3 and 5. Okay, so let's write a function to plot pixels in mode 4.

```
void DrawPixel4(int x, int y, unsigned char color)
{
    unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;

    pixel = videoBuffer[offset];
    if (x & 1)
        videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
```

```
videoBuffer[offset] = (pixel & 0xFF00) + color;
```

}

The Mode4Pixels program uses the standard library (stdlib.h) to gain access to a pseudo-random number generator in order to plot random pixels on the screen. This is the same rand() function you have seen in earlier programs in this chapter. The source code for the Mode4Pixels program should be helpful to you if you ever plan to write a game using mode 4, because the DrawPixel4 function can be very helpful indeed. I somewhat dislike mode 4 because of the packed pixels, but it does have advantages, as I have explained already. Here is the source code for Mode4Pixels.

```
///////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Pixels Project
// main.c source code file
///////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//add support for the rand function
#include <stdlib.h>


//declare the function prototype
void DrawPixel4(int x, int y, unsigned char bColor);


//declare some defines for the video mode
#define MODE_4 0x4
#define BG2_ENABLE 0x400
#define REG_DISPCNT *(unsigned int*)0x4000000
#define RGB(r,g,b)  (unsigned short)((r)+((g)<<5)+((b)<<10))


//create a pointer to the video and palette buffers
unsigned short* videoBuffer = (unsigned short*)0x6000000;
```

```
unsigned short* paletteMem = (unsigned short*)0x5000000;


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x1,y1,n;

    SetMode(MODE_4 | BG2_ENABLE);

    for (n = 1; n < 256; n++)
        paletteMem[n] = RGB(rand() % 31, rand() % 31, rand() % 31);


    while(1)
    {
        x1 = rand() % 240;
        y1 = rand() % 160;

        DrawPixel4(x1, y1, rand() % 256);
    }


    return 0;
}



/////////////////////////////////////////////////////////////
// Function: DrawPixel4
// Draws a pixel in mode 4
/////////////////////////////////////////////////////////////
void DrawPixel4(int x, int y, unsigned char color)
```

```
{
    unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;


    pixel = videoBuffer[offset];
    if (x & 1)
        videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
        videoBuffer[offset] = (pixel & 0xFF00) + color;
}
```

The output from the Mode4Pixels program is shown in Figure 5.10.
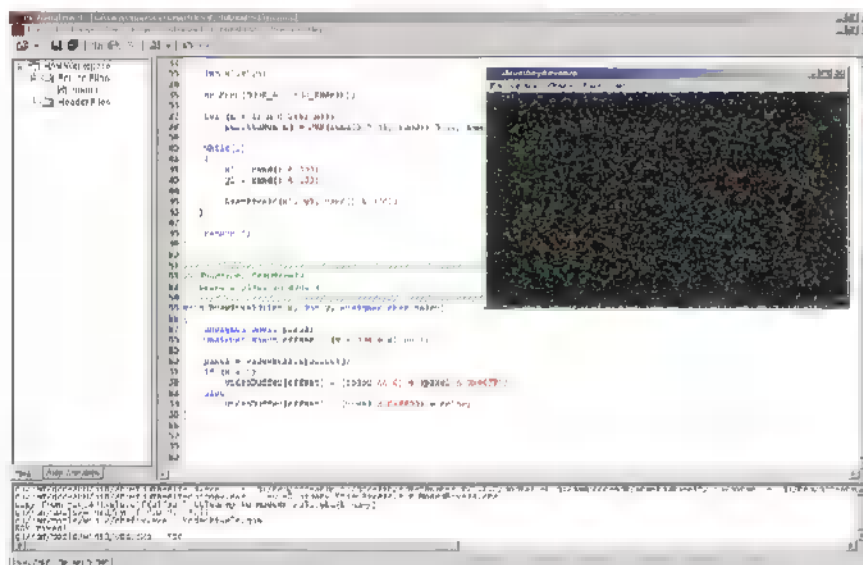
Figure 5.10
The Mode4Pixels
program draws
random pixels on
the GBA screen.

Using this new DrawPixel4 function and the sample program, you should be able to modify the mode 3 samples for drawing lines, circles, and filled boxes. I won't waste space by listing programs when the only difference is in the DrawPixel function (DrawPixel3, DrawPixel4, and DrawPixel5).. But I encourage you to plug DrawPixel4 into those projects to see how well it works.

Drawing Bitmaps

Since mode 4 is an 8-bit mode (as well you know at this point), you can't use the same file generated by gfx2gba for mode 4 that you used for mode 3. For one thing, there are only 38,400 bytes in a mode 4 image (that is, a full-screen background image), while mode 3

backgrounds are 76,800 bytes (exactly twice the size). Let me show you how to convert a bitmap that is similar to the mode3.bmp file. This time, I'll call the new file mode4.bmp. Just use your favorite graphics editing program to create a new 240 x 160 image, and make sure it's 8-bit, with 256 colors, the only format supported by gfx2gba. If you prefer, you may look at the project folder on the CD-ROM under \Sources\Chapter05\Mode4Bitmap. Here's the command to convert the bitmap to a C array:

```
gfx2gba -fsrc -pmode4.pal mode4.bmp
```

These parameters for gfx2gba generate two files: mode4.raw.c and mode4.pal.c, containing the bitmap and palette, respectively. Setting up the palette for a bitmap is similar to setting it up for drawing pixels, except that now you use the mode4_Palette array instead of random numbers. I've written another program to show how to display a bitmap in mode 4. This project is called Mode4Bitmap and is similar to the Mode3Bitmap program. Just create a new project in Visual HAM and replace all the default code in main.c with this code and run it by pressing F7.

```
/////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Bitmap Project
// main.c source code file
/////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//include files
#include <stdlib.h>
#include <string.h>
#include "mode4.raw.c"
#include "mode4.pal.c"

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_4 0x4
```

```c
#define BG2_ENABLE 0x400


//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)


//create a pointer to the video and palette buffers
unsigned short* videoBuffer = (unsigned short*)0x6000000;
unsigned short* paletteMem = (unsigned short*)0x5000000;


//////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////
int main(void)
{
    int n;

    //set video mode 4
    SetMode(MODE_4 | BG2_ENABLE);

    //set up the palette colors
    for (n = 0; n < 256; n++)
        paletteMem[n] = mode4_Palette[n];

    //display the bitmap
    memcpy(videoBuffer, mode4_Bitmap, 38400);

    //endless loop
    while(1)
    {
    }

    return 0;
}
```

Whoa, wait a second! This program isn't like Mode3Bitmap at all. Where's the DrawPixel4 function? I'm sure you noticed that memcpy function (which was the reason this program needed to include string.h). The memcpy function copies a specified number of bytes from a source buffer to a destination buffer. It works great for displaying a bitmap in mode 4! Figure 5.11 shows the output from the program. This ANSI C function works in exactly the same manner as it does on other systems. In fact, the same can be said about all of the C libraries included with the HAM SDK, as all are ANSI C compliant and exactly the same functions that you will find on any platform. This is good news for cross-platform development!
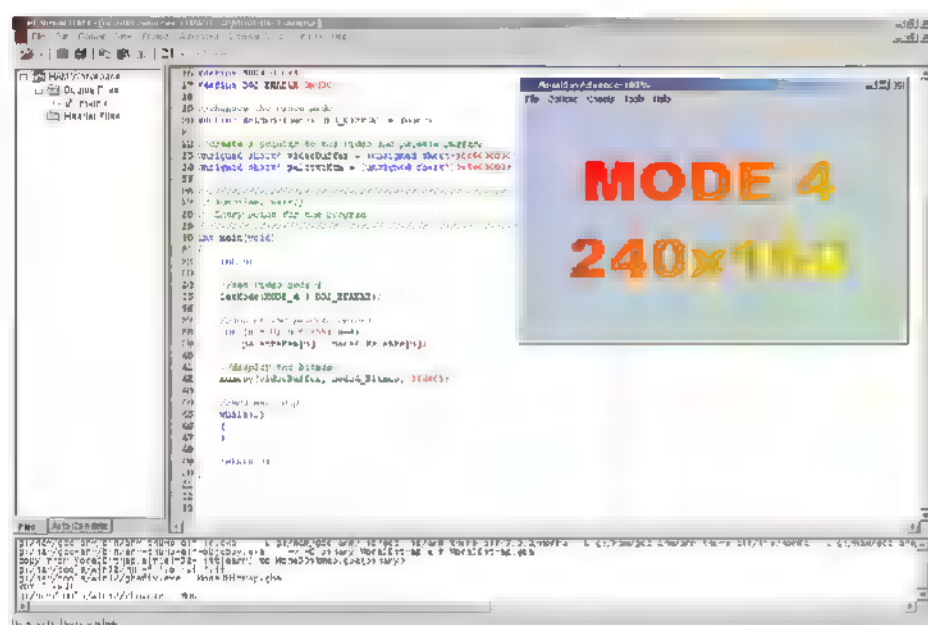
Figure 5.11
The Mode4Bitmap program shows how to display a bitmap in mode 4 using memcpy.

There are other ways to display a bitmap too. Memcpy may not be the fastest method, because it doesn't take advantage of writing two pixels at a time—it just copies one whole buffer to the screen. However, it might be faster to copy mode4_Bitmap to the videoBuffer using a loop that iterates 120 x 160 times (that's half the number of pixels). You may try this if you wish, but I'm not going to get into optimization at this point because DMA is faster than any software loop.

Page Flipping and Double Buffering

Page flipping is built into modes 4 and 5, so I'll show you how to use this great feature. Basically, when you draw everything into an off-screen buffer, things move along much more quickly. In fact, using a double buffer makes the drawing operations so fast that it

interferes with the vertical refresh, so you must add code to check for the vertical blank period and only flip the page during this period.

In order to do page flipping, the program needs two video buffers instead of one—the front buffer and the rear buffer. These take up the same amount of video memory as the mode 3 buffer, and like I said, this is all part of the architecture of mode 4. The front buffer is located at 0 x 6000000 like usual, while the back buffer is located at 0x600A000. Here are the definitions:

```
unsigned short* FrontBuffer = (unsigned short*)0x6000000;
unsigned short* BackBuffer = (unsigned short*)0x600A000;
```

Using a back buffer bit modifier:

```
#define BACKBUFFER 0x10
```

along with the video mode register, we can write a function to flip from the front to the back buffer by simply changing the video buffer pointer. This simply redirects the GBA from one buffer to the other automatically—without requiring you to copy pixels!

```
void FlipPage(void)
{
    if(REG_DISPCNT & BACKBUFFER)
    {
        REG_DISPCNT &= ~BACKBUFFER;
        videoBuffer = BackBuffer;
    }
    else
    {
        REG_DISPCNT |= BACKBUFFER;
        videoBuffer = FrontBuffer;
    }
}
```

Here is the source code listing for the Mode4Flip program. You may open this project directly off the CD-ROM, located in \Sources\Chapter05\Mode4Flip, although all of these programs are short enough to be simply typed into Visual HAM and run directly. The exception arises when using media resources (bitmaps and waves), at which point you may

want to copy the converted media files if you are still not fluent in the process of converting files to the GBA yet.

```c
/////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Flip Project
// main.c source code file
/////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>
#include <string.h>

//declare the function prototype
void DrawPixel4(int, int, unsigned char);
void DrawBox4(int, int, int, int, unsigned char);
void FlipPage(void);
void WaitVBlank(void);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_4 0x4
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) (unsigned short)((r)+((g)<<5)+((b)<<10))

//video buffer defines
```

```
#define BACKBUFFER 0x10

unsigned short* FrontBuffer = (unsigned short*)0x6000000;

unsigned short* BackBuffer = (unsigned short*)0x600A000;

unsigned short* videoBuffer;

unsigned short* paletteMem = (unsigned short*)0x5000000;


volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;


/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int n;

    //set video mode and start page flipping
    SetMode(MODE_4 | BG2_ENABLE);
    FlipPage();

    //set the first two palette entries
    paletteMem = RGB(0, 31, 0);
    paletteMem = RGB(31, 0, 0);

    //draw the first random box
    DrawBox4(20, 20, 100, 140, 1);

    //flip the page to the back buffer
    FlipPage();

    //draw the second random box
    DrawBox4(140, 20, 220, 140, 2);
```

```
            while(1)
            {
                //wait for vertical blank
                WaitVBlank();

                //flip the page
                FlipPage();

                //slow it down--modify as needed
                n = 500000;
                while(n--);
            }

            return 0;
}


////////////////////////////////////////////////////////////
// Function: DrawPixel4
// Draws a pixel in mode 4
////////////////////////////////////////////////////////////
void DrawPixel4(int x, int y, unsigned char color)
{
    unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;

    pixel = videoBuffer[offset];
    if (x & 1)
        videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
        videoBuffer[offset] = (pixel & 0xFF00) + color;
}


////////////////////////////////////////////////////////////
```

```
// Function: DrawBox4
// Draws a filled box
////////////////////////////////////////////////////////
void DrawBox4(int left, int top, int right, int bottom,
    unsigned char color)
{
    int x, y;

    for(y = top; y < bottom; y++)
        for(x = left; x < right; x++)
            DrawPixel4(x, y, color);
}


////////////////////////////////////////////////////////
// Function: FlipPage
// Switches between the front and back buffers
////////////////////////////////////////////////////////
void FlipPage(void)
{
    if(REG_DISPCNT & BACKBUFFER)
    {
        REG_DISPCNT &= ~BACKBUFFER;
        videoBuffer = BackBuffer;
    }
    else
    {
        REG_DISPCNT |= BACKBUFFER;
        videoBuffer = FrontBuffer;
    }
}
////////////////////////////////////////////////////////
// Function: WaitVBlank
```

```
// Checks the scanline counter for the vertical blank period
//////////////////////////////////////////////////////////////
void WaitVBlank(void)

{

    while(*ScanlineCounter < 160);


}
```

The output from the Mode4Flip program is shown in Figure 5.12.
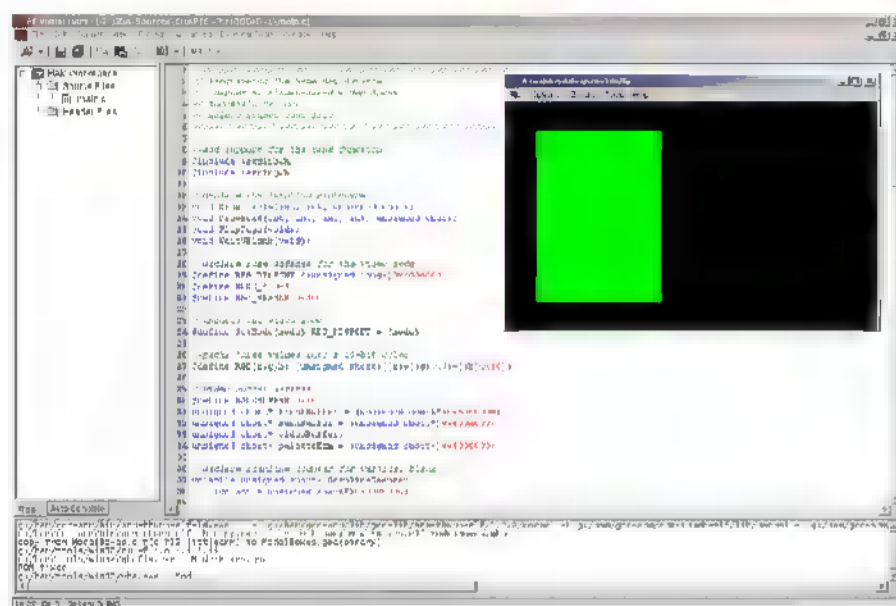
Figure 5.12
The Mode4Flip program
demonstrates how page
flipping works.

Working with Mode 5

Mode 5 is similar to mode 3 in that it features 15-bit pixels and therefore has no need for a
palette. However, mode 5 has a lower resolution than either of the previous two modes:
only 160 x 128! That is much smaller than 240 x 160, but it's a compromise in that you get a
double buffer and also 15-bit color.

Drawing Pixels

Mode 5 is so similar to mode 3 that it doesn't require a lengthy explanation. Basically, you
can just modify the DrawPixel3 function so that it takes into account the more limited
screen resolution of mode 5 and come up with a new function:

```
void DrawPixel5(int x, int y, unsigned short c)
```

```
{

    videoBuffer[y * 160 + x] = c;

}
```

Testing Mode 5

The Mode5Pixels program is similar to the pixel program for mode 3. Basically, just change the SetMode line so it uses MODE_5 and change the range of the random numbers to take into account the 160 x 128 resolution. Drawing bitmaps is precisely the same for mode 5 as it is for mode 3, the only exception being the smaller size. If you want to write ■ bitmap display program for mode 5, just create a bitmap image that is 160 x 128 pixels in size, then run gfx2gba using the same parameters for mode 3, and use mode5_Bitmap instead of mode3_Bitmap. Simple! Here is the complete Mode5Pixels program:

```
///////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode5Pixels Project
// main.c source code file
///////////////////////////////////////////////////////////

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel5(int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_5 0x5
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```c
//packs three values into a 15-bit color
#define RGB(r,g,b)  ((r)+(g<<5)+(b<<10))


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


/////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_5 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
    x = rand() % 160;
        y = rand() % 128;
        color = RGB(rand()%31, rand()%31, rand()%31);
        DrawPixel5(x, y, color);
    }

    return 0;
}


/////////////////////////////////////////////////////////
// Function: DrawPixel5
// Draws a pixel in mode 5
/////////////////////////////////////////////////////////
void DrawPixel5(int x, int y, unsigned short c)
```

```
{
    videoBuffer[y * 160 + x] = c;
}
```

The output from the Mode5Pixels program is shown in Figure 5.13.
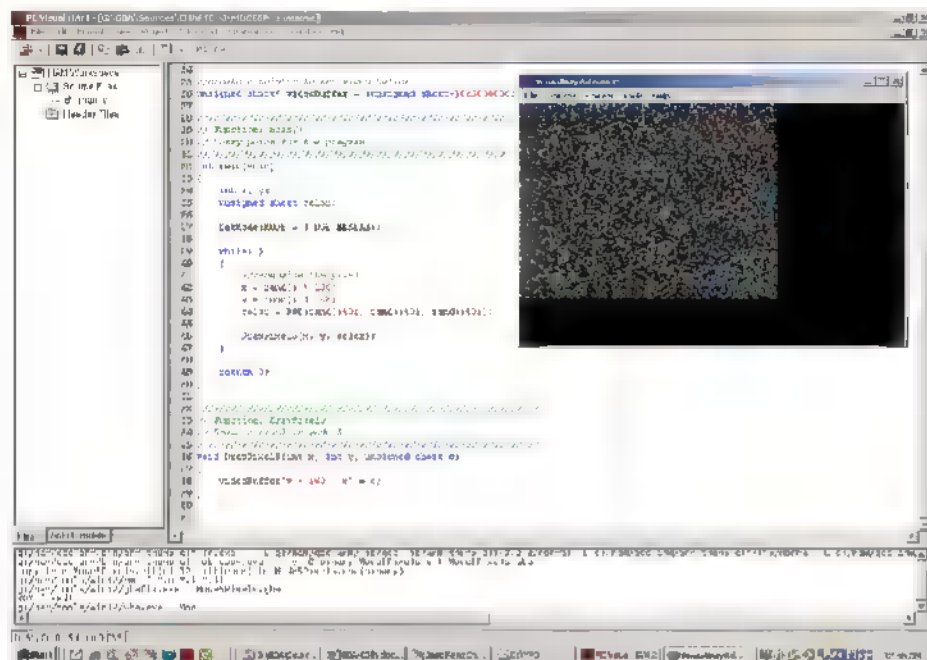
Figure 5.13
The Mode5Pixels
program draws
random pixels on
the GBA screen.

Printing Text on the Screen

At this point I believe we've conquered the bitmapped video modes on the GBA, so I'd like to talk about a very important subject that is directly related to the subjects already covered, and that is text output. You will almost immediately find a need to display text on the screen, in order to present the player with a game menu, to display messages on the screen . . . whatever! Text output is not built into the GBA, although it is a feature available in Hamlib (as you saw back in Chapter 4, "Starting with the Basics" with the Greeting program).

Text output must be done the hard way, just like drawing lines, circles, and boxes; that is, you must write code to display the pixels of each character in a font that you must create from scratch. Now, I realize that many programmers use a bitmapped font, and that's not a bad idea at all, because the font characters can be treated as sprites. However, I prefer a low-memory footprint and more control over the font display mechanism. I have written two functions to display the font (which I will cover shortly). The Print function accepts a location, string, and color for the text output. The DrawChar function actually does the

work of drawing each pixel in the font character, which is passed one at a time from the Print function.

```
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color);
        pos += 8;
    }
}


void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            // grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            // if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
        }
}
```

The Hard-Coded Font

Now, to make sense of these text output functions, you'll first need a font. I have created a font for this purpose; it is just a large array of numbers. To make the font source easier to read, I defined the letter W to represent 1, which really helps when typing in the code. Of course, you may grab this font.h file from the CD-ROM. It is located in

\Sources\Chapter05\DrawText. If you are typing in the code, you'll want to create a new project in Visual HAM called DrawText and add a new file to the project called font.h, which you may type the following code into. This font file will be used by nearly every sample program from this point forward.

```
#ifndef _FONT_H
#define _FONT_H

#define W 1

unsigned short font[] =
{
// (space) 32
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// ! 33
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
// " 34
    0,0,0,0,0,0,0,0,
    0,W,W,0,W,W,0,0,
    0,W,W,0,W,W,0,0,
    0,0,W,0,0,W,0,0,
```

```
    0,0,W,0,0,W,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// # 35
    0,0,0,0,0,0,0,0,
    0,0,W,0,0,W,0,0,
    0,W,W,W,W,W,W,0,
    0,0,W,0,0,W,0,0,
    0,0,W,0,0,W,0,0,
    0,W,W,W,W,W,W,0,
    0,0,W,0,0,W,0,0,
    0,0,0,0,0,0,0,0,
// $ 36
    0,0,0,W,0,0,0,0,
    0,0,W,W,W,W,0,0,
    0,W,0,W,0,0,0,0,
    0,0,W,W,W,0,0,0,
    0,0,0,W,0,W,0,0,
    0,W,W,W,W,0,0,0,
    0,0,0,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
// % 37
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,W,0,0,0,
    0,0,0,W,0,0,0,0,
    0,0,W,0,W,W,0,0,
    0,W,0,0,W,W,0,0,
    0,0,0,0,0,0,0,0,
// & 38
    0,0,0,0,0,0,0,0,
    0,0,W,W,W,0,0,0,
```

```
        0,w,0,0,0,w,0,0,
        0,0,w,w,w,0,0,0,
        0,w,0,w,0,0,0,0,
        0,w,0,0,w,0,w,0,
        0,0,w,w,w,w,0,0,
        0,0,0,0,0,0,w,0,
// '  39
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,0,w,0,0,0,
        0,0,0,0,w,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
// (  40
        0,0,0,0,0,0,w,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,w,0,0,
        0,0,0,0,0,0,w,0,
// )  41
        0,w,0,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,0,w,0,0,0,0,0,
        0,w,0,0,0,0,0,0,
// *  42
```

```
        0,0,0,0,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,W,0,W,0,W,0,0,
        0,0,W,W,W,0,0,0,
        0,0,W,W,W,0,0,0,
        0,0,W,W,W,0,0,0,
        0,W,0,0,0,W,0,0,
        0,0,0,0,0,0,0,0,
// + 43
        0,0,0,0,0,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,W,W,W,W,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,0,0,0,0,0,
// , 44
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,W,W,0,0,0,0,
        0,0,W,0,0,0,0,0,
// - 45
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,W,W,W,W,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
```

```
        0,0,0,0,0,0,0,0,
// . 46
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,w,w,0,0,0,0,
        0,0,w,w,0,0,0,0,
        0,0,0,0,0,0,0,0,
// / 47
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,w,0,
        0,0,0,0,0,w,w,0,
        0,0,0,0,w,w,0,0,
        0,0,0,w,w,0,0,0,
        0,0,w,w,0,0,0,0,
        0,w,w,0,0,0,0,0,
        0,w,0,0,0,0,0,0,
// 0 48
        0,0,w,w,w,w,0,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,w,w,0,
        0,w,w,0,w,0,w,0,
        0,w,w,w,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,0,w,w,w,w,0,0,
// 1 49
        0,0,0,0,w,0,0,0,
        0,0,w,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
```

```
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,W,W,W,W,0,0,
// 2 50
        0,0,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,0,0,0,0,W,0,
        0,0,0,0,0,W,W,0,
        0,0,0,0,W,W,0,0,
        0,0,0,W,W,0,0,0,
        0,0,W,W,0,0,0,0,
        0,W,W,W,W,W,W,0,
// 3 51
        0,0,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,0,0,0,0,W,0,
        0,0,0,0,0,0,W,0,
        0,0,0,W,W,W,0,0,
        0,0,0,0,0,0,W,0,
        0,W,0,0,0,0,W,0,
        0,0,W,W,W,W,0,0,
// 4 52
        0,0,0,0,W,W,0,0,
        0,0,0,W,W,W,0,0,
        0,0,W,W,0,W,0,0,
        0,W,W,0,0,W,0,0,
        0,W,W,W,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
// 5 53
        0,W,W,W,W,W,W,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,0,0,
```

```
        0,w,w,0,0,0,0,0,
        0,0,w,w,w,w,0,0,
        0,0,0,0,0,0,w,0,
        0,w,0,0,0,0,w,0,
        0,0,w,w,w,w,0,0,
// 6 54
        0,0,0,w,w,w,w,0,
        0,0,w,w,w,w,w,0,
        0,w,w,0,0,0,0,0,
        0,w,w,0,0,0,0,0,
        0,w,w,w,w,w,0,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,0,w,w,w,w,0,0,
// 7 55
        0,w,w,w,w,w,w,0,
        0,w,w,w,w,w,w,0,
        0,w,0,0,0,w,w,0,
        0,0,0,0,w,w,0,0,
        0,0,0,0,w,w,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,w,w,0,0,0,0,
// 8 56
        0,0,w,w,w,w,0,0,
        0,w,w,w,w,w,w,0,
        0,w,w,0,0,0,w,0,
        0,0,w,0,0,0,w,0,
        0,0,w,w,w,w,0,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,0,w,w,w,w,0,0,
// 9 57
        0,0,w,w,w,w,0,0,
```

```
        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,0,W,W,W,W,W,0,

        0,0,0,0,0,W,W,0,

        0,0,0,0,0,W,W,0,

        0,0,0,0,0,W,W,0,

    // : 58

        0,0,0,0,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,0,0,0,0,0,0,

        0,0,0,0,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,0,0,0,0,0,0,

    // ; 59

        0,0,W,W,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,0,0,0,0,0,0,

        0,0,0,0,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

    // < 60

        0,0,0,0,W,W,0,0,

        0,0,0,W,W,0,0,0,

        0,0,W,W,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,0,W,W,0,0,0,0,

        0,0,0,W,W,0,0,0,

        0,0,0,0,W,W,0,0,
```

```
// = 61
    0,0,0,0,0,0,0,0,
    0,w,w,w,w,w,0,0,
    0,w,w,w,w,w,0,0,
    0,0,0,0,0,0,0,0,
    0,w,w,w,w,w,0,0,
    0,w,w,w,w,w,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// > 62
    0,w,w,0,0,0,0,0,
    0,0,w,w,0,0,0,0,
    0,0,0,w,w,0,0,0,
    0,0,0,0,w,w,0,0,
    0,0,0,0,w,w,0,0,
    0,0,0,w,w,0,0,0,
    0,0,w,w,0,0,0,0,
    0,w,w,0,0,0,0,0,
// ? 63
    0,0,w,w,w,w,0,0,
    0,w,w,w,w,w,w,0,
    0,w,0,0,0,w,w,0,
    0,0,0,0,w,w,0,0,
    0,0,0,w,w,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,w,w,0,0,0,
    0,0,0,w,w,0,0,0,
// @ 64
    0,0,0,0,0,0,0,0,
    0,0,w,w,w,0,0,0,
    0,w,0,0,0,w,0,0,
    0,w,0,w,w,0,0,0,
    0,w,0,w,w,0,0,0,
    0,w,0,0,0,0,w,0,
```

```
    0,0,W,W,W,W,0,0,
    0,0,0,0,0,0,0,0,

// A 65
    0,0,W,W,W,0,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,0,W,0,0,

// B 66
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,0,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,W,W,W,0,0,

// C 67
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,

// D 68
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
```

```
        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,W,W,W,0,0,

// E 69

        0,W,W,W,W,W,W,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,W,W,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,W,W,W,W,0,

// F 70

        0,W,W,W,W,W,W,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,W,W,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

// G 71

        0,0,W,W,W,W,0,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,0,W,W,W,W,0,0,

// H 72

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,
```

```
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,w,w,w,w,w,w,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,0,w,0,
// I 73
        0,0,w,w,w,w,0,0,
        0,0,w,w,w,w,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,0,w,w,0,0,0,
        0,0,w,w,w,w,0,0,
// J 74
        0,0,0,0,w,w,0,0,
        0,0,0,0,w,w,0,0,
        0,0,0,0,w,w,0,0,
        0,0,0,0,w,w,0,0,
        0,0,0,0,w,w,0,0,
        0,w,0,0,w,w,0,0,
        0,w,w,w,w,w,0,0,
        0,0,w,w,w,0,0,0,
// K 75
        0,w,w,0,0,0,w,0,
        0,w,w,0,0,w,w,0,
        0,w,w,0,w,w,0,0,
        0,w,w,w,w,0,0,0,
        0,w,w,w,w,0,0,0,
        0,w,w,0,w,w,0,0,
        0,w,w,0,0,w,w,0,
        0,w,w,0,0,0,w,0,
// L 76
```

```
        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,O,O,O,O,O,

        O,W,W,W,W,W,W,O,

    // M 77

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,W,O,W,W,O,

        O,W,W,W,W,W,W,O,

        O,W,W,O,W,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

    // N 78

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,W,O,O,W,O,

        O,W,W,W,W,O,W,O,

        O,W,W,O,W,W,W,O,

        O,W,W,O,O,W,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

    // O 79

        O,O,W,W,W,W,O,O,

        O,W,W,W,W,W,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,

        O,W,W,O,O,O,W,O,
```

```
        0,0,W,W,W,W,0,0,

    // P  80

        0,W,W,W,W,W,0,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,W,W,W,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

        0,W,W,0,0,0,0,0,

    // Q  81

        0,0,W,W,W,W,0,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,W,W,0,

        0,W,W,0,0,0,W,0,

        0,0,W,W,W,W,0,W,

    // ▉  82

        0,W,W,W,W,W,0,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,W,0,0,

        0,W,W,W,W,0,0,0,

        0,W,W,0,W,W,0,0,

        0,W,W,0,0,W,W,0,

    // S  83

        0,0,W,W,W,W,0,0,

        0,W,W,W,W,W,W,0,

        0,W,W,0,0,0,W,0,

        0,W,W,0,0,0,0,0,

        0,0,W,W,W,W,0,0,
```

```
        O,O,O,O,O,O,w,O,
        O,w,O,O,O,O,w,O,
        O,O,w,w,w,w,O,O,

// T 84
        O,w,w,w,w,w,w,O,
        O,w,w,w,w,w,w,O,
        O,O,O,w,w,O,O,O,
        O,O,O,w,w,O,O,O,
        O,O,O,w,w,O,O,O,
        O,O,O,w,w,O,O,O,
        O,O,O,w,w,O,O,O,
        O,O,O,w,w,O,O,O,

// U 85
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,O,w,w,w,w,O,O,

// V 86
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,w,w,O,O,O,w,O,
        O,O,w,w,O,w,O,O,
        O,O,w,w,O,w,O,O,
        O,O,w,w,O,w,O,O,
        O,O,O,w,w,O,O,O,

// W 87
        O,w,w,O,O,O,O,w,
        O,w,w,O,O,O,O,w,
        O,w,w,O,O,O,O,w,
```

```
        O,W,W,O,O,O,O,W,
        O,W,W,O,W,W,O,W,
        O,O,W,W,O,O,W,O,
        O,O,W,W,O,O,W,O,
        O,O,W,W,O,O,W,O,
// X 88
        O,W,W,O,O,O,W,O,
        O,W,W,O,O,O,W,O,
        O,O,W,W,O,W,O,O,
        O,O,O,W,W,O,O,O,
        O,O,O,W,W,O,O,O,
        O,O,W,W,O,W,O,O,
        O,W,W,O,O,O,W,O,
        O,W,W,O,O,O,W,O,
// Y 89
        O,W,W,O,O,O,W,O,
        O,W,W,O,O,O,W,O,
        O,W,W,O,O,O,W,O,
        O,O,W,W,W,W,O,O,
        O,O,O,W,W,O,O,O,
        O,O,O,W,W,O,O,O,
        O,O,O,W,W,O,O,O,
        O,O,O,W,W,O,O,O,
// Z 90
        O,W,W,W,W,W,W,O,
        O,W,W,W,W,W,W,O,
        O,O,O,O,O,W,W,O,
        O,O,O,O,W,W,O,O,
        O,O,O,W,W,O,O,O,
        O,O,W,W,O,O,O,O,
        O,W,W,O,O,O,O,O,
        O,W,W,W,W,W,W,O,
// [ 91
        O,O,O,O,W,W,W,O,
```

```
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,W,0,

// \  92
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,0,W,W,0,0,0,0,
        0,0,W,W,0,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,

// ]  93
        0,W,W,W,0,0,0,0,
        0,W,W,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,W,W,W,0,0,0,0,
};
#endif
```

The DrawText Program

The DrawText program uses mode 3 to test the hard-coded font that you typed into the
font.h file. Create a new project called DrawText and copy the font.h file into the folder
for this new project. Following is the source code for the complete DrawText program that
demonstrates the new font. If you would like to test this program using modes 4 or 5, you

will need to change the DrawChar function so it calls DrawPixel4 or DrawPixel5, respectively, and then be sure to include those functions in the program.

```c
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// DrawText Project
// main.c source code file
//////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "font.h"

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);

//create some color constants
#define WHITE 0xFFFF
#define RED 0x00FF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define some video mode values
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
```

```c
unsigned short* videoBuffer = (unsigned short*)0x6000000;


////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////
int main()
{
    char *test = "TESTING...1...2...3...";
    int pos = 0;

    //switch to video mode 3 (240x160 16-bit)
    REG_DISPCNT = (MODE_3 | BG2_ENABLE);

    Print(1, 1, "DRAWTEXT PROGRAM", RED);
    Print(1, 20, "()*+,-.0123456789:;<=>?@", GREEN);
    Print(1, 30, "ABCDEFGHIJKLMNOPQRSTUVWXYZ[/]", BLUE);
    Print(1, 50, "BITMAP FONTS ARE A CINCH!", MAGENTA);
    Print(1, 60, "(JUST BE SURE TO USE CAPS)", CYAN);

    //display each character in a different color
    while (*test)
    {
        DrawChar(1 + pos, 80, *test++, 0xBB + pos * 16);
        pos += 8;
    }

    Print(1, 100, "THAT'S ALL, FOLKS =]", BROWN);

    //continuous loop
    while(1)
    {
    }
```

```
        return 0;
    }



    ///////////////////////////////////////////////
    // Function: DrawPixel3
    // Draws a pixel in mode 3
    ///////////////////////////////////////////////
    void DrawPixel3(int x, int y, unsigned short color)
    {
        videoBuffer[y * 240 + x] = color;
    }



    ///////////////////////////////////////////////
    // Function: Print
    // Prints a string using the hard-coded font
    ///////////////////////////////////////////////
    void Print(int left, int top, char *str, unsigned short color)
    {
        int pos = 0;
        while (*str)
        {
            DrawChar(left + pos, top, *str++, color);
            pos += 8;
        }
    }



    ///////////////////////////////////////////////
    // Function: DrawChar
    // Draws a character one pixel at a time
    ///////////////////////////////////////////////
    void DrawChar(int left, int top, char letter, unsigned short color)
    {
        int x, y;
        int draw;
```

```
for(y = 0; y < 8; y++)

      for (x = 0; x < 8; x++)

      {

            // grab a pixel from the font char

      draw = font[(letter-32) * 64 + y * 8 + x];

      // if pixel = 1, then draw it

      if (draw)

            DrawPixel3(left + x, top + y, color);

      }

}
```

The output from the DrawText program is shown in Figure 5.14.
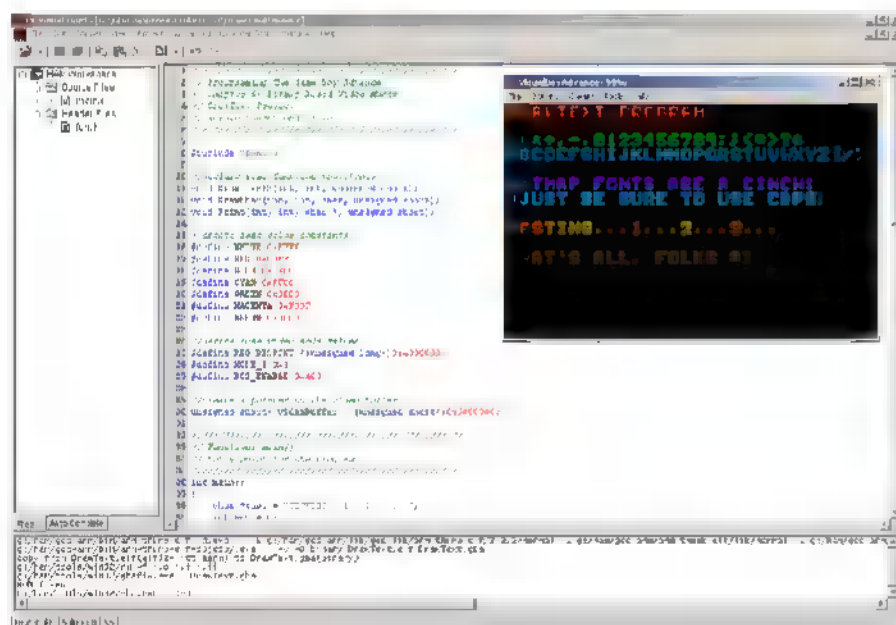
Figure 5.14

The DrawText program uses the hard-coded font to print messages on the GBA screen.

Summary

This chapter conquered the extremely complex subject of doing graphics on the GBA using bitmapped video modes. Along the way, you learned about modes 3, 4, and 5 and how to draw pixels, lines, and filled boxes. This chapter also showed how to convert bitmap images to GBA format and then display them on the screen using either 8-bit mode 4 or 15-bit mode 3. Learning the ropes when it comes to graphics on a console is really the key to everything else in the games you are likely to write, because without a basic understanding of the graphics system, you will be unable to get even a simple Pong-style game on the

screen. This chapter concluded by providing a useful font and functions for displaying text on the screen.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: Mode 5 was somewhat neglected in this chapter, but only because it is very similar to mode 3, and therefore I felt it would be repetitive to provide additional examples when only a single line of code is at stake. Now it's up to you! Write a program that displays a bitmap image using mode 5.

Challenge 2: The Mode3Boxes program could use some optimization. Now that you know how effective the memcpy and memset functions can be, this is a good opportunity to speed up the program. Modify the DrawBox3 function so it uses memcpy to fill a line of pixels on the screen, thus replacing the x loop. All you need to do is determine the number of x pixels across to use as the count for memset, and of course the color is used for the value parameter.

Challenge 3: Now for a real challenge! Add the font.h file and text functions to one of the mode 3 example programs, and then display the number of objects that have been drawn on the screen. If you want to add insult to injury, add the lowercase letters to the font so the Print function will be able to draw more of the ASCII character set.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. What is the memory address for the video buffer?
   - A. ▯ x 7006000
   - B. 0 x 6000000
   - C. 0 x 6005000
   - D. ▯ x 4928300

2. Which video mode features a resolution of 240 x 160 and also a double buffer?

    A. Mode 4

    B. Mode 3

    C. Mode 5

    D. Mode 2

3. True or False: Video mode 5 uses a color palette.

    A. True

    B. False

4. Which video mode has a resolution of 160 x 128?

    A. Mode 6

    B. Mode 4

    C. Mode 5

    D. Mode 3

5. What is the address of the back buffer in mode 4?

    A. 0 x 6000000

    B. 0 x 7006000

    C. 0 x 6005000

    D. 0 x 600A000

6. What is the name of the line-drawing algorithm used in the Mode3Lines program?

    A. Einstein

    B. Bresenham

    C. Hawking

    D. Von Neumann

7. Which video mode has a resolution of 240 x 160, 15-bit color, and no back buffer?

    A. Mode 3

    B. Mode 2

    C. Mode 5

    D. Mode 4

8. What data type is used to reference the 15-bit video buffer for modes 3 and 5?

    A. unsigned char

    B. unsigned int

C. unsigned short

D. unsigned check

9. What is the color depth of the display in video mode 4?

   A. 32 bits

   B. 16 bits

   C. 64 bits

   D. 8 bits

10. What is the name of the organization that linked thousands of Game Boy Advance units together in order to create a supercomputer?

   A. GBA-SETI

   B. GBA White

   C. Pocket Cray

   D. I don't think so!

Chapter 6

Tile-Based

Video Modes

This chapter explains the Game Boy Advance's tile-based graphics modes, with coverage of tile images, tile maps, scrolling backgrounds, and rotating backgrounds, as well as a tutorial on creating tiles, and converting them to a C array. Two complete programs are included in this chapter to demonstrate how to use scrolling and rotating backgrounds: the TileMode0 program and the RotateMode2 program. Here are the key topics covered:

- Introduction to tile-based video modes
- Creating a scrolling background
- Creating a rotating background

Introduction to Tile-Based Video Modes

The Game Boy Advance offers three tile-based (also called text-based, or character) video modes that support a tiled screen comprising 8 x 8 tiles. A full screen is therefore made up of 30 tiles across and 20 tiles down. The maximum size of the background tile map is 1024 x 1024 pixels (when a rotation map is being used, 512 x 512 otherwise), or rather, 128 tiles across and 128 tiles down. As you might imagine, this provides the capability for storing a sizable level in a single tile map. Table 6.1 shows the properties of the tile-based video modes.

Table 6.1 Tiled Video Modes

Mode	Backgrounds	Rotation/Scaling
0	0, 1, 2, 3	No
1	0, 1, 2	Yes (2)
2	2, 3	Yes (2, 3)

For instance, six rows or six columns of tiles can be stored in the 128 x 128 tile map and scrolled horizontally or vertically, adjusting the position of the "screen" to the next row or column upon reaching the edge of the previous one. See Figure 6.1.
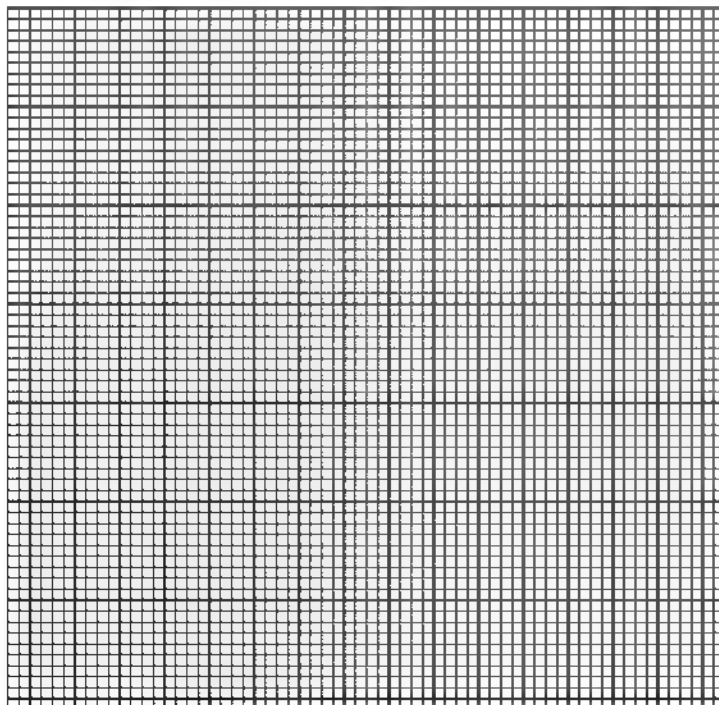
Figure 6.1
The maximum size of a
(non-rotation) tile map
is 512 x 512 pixels, or
64 x 64 tiles.

As you can see from Figure 6.2, you can create a large tile map for a game indeed, because the image in the upper-left corner represents one full screen of tiles!
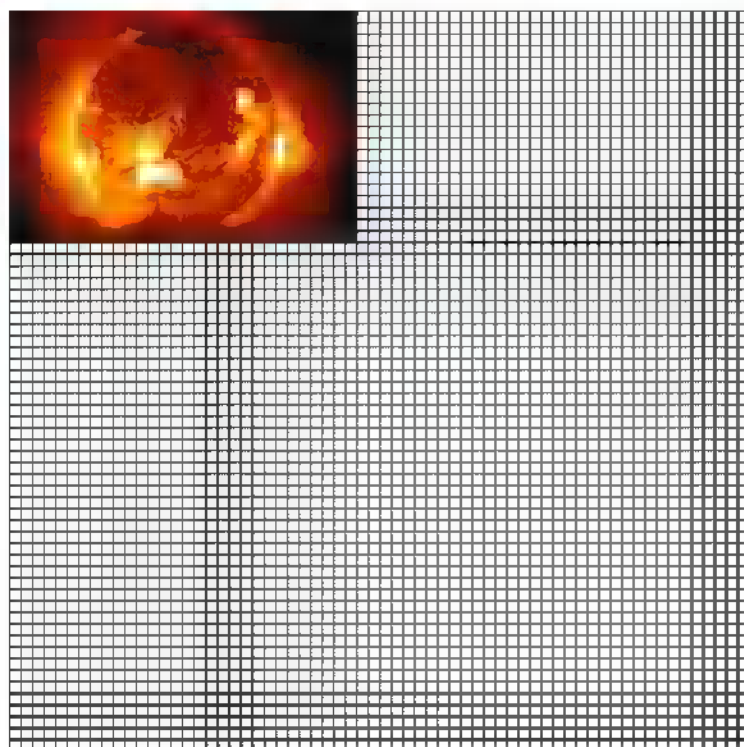
Figure 6.2

A single screen uses only a small portion of the maximum number of tiles.

Backgrounds

Since the tiled "text" backgrounds (0 and 1) support hardware scrolling of the background, you can simply plug in all the tiles you need for a game level (which would obviously not be made up of a single picture as shown in Figure 6.2). A typical tile-based game will have hundreds of tiles, many of which make up larger tiles (such as 16 x 16, 32 x 32, and 64 x 64 or larger). Displaying a larger "tile" really just involves displaying the smaller tiles that make up that large tile. It all breaks down to the least common denominator, which is the 8 x 8 pixel tile.

On the other hand, there are the two scale and rotate backgrounds (2 and 3). These backgrounds support only 8-bit color and vary in resolution from 128 to 1,024 pixels across. The usual palette located at 0 x 5000000 contains 256 color values, each a 16-bit number (which you learned about in the last chapter). I should also point out that the tile-based modes support 16-color palettes, and when using 16-color palettes, there are 16 separate, individual palettes available. Due to the smaller memory footprint of a 4-bit color (one-fourth the size of a 16-bit color), images that use 16-color palettes are also smaller. I will be sticking to 8-bit and 16-bit (actually, 15-bit, as you have already learned) colors for

simplicity. As Table 6.2 shows, backgrounds ▇ and 1 are text backgrounds, while backgrounds 2 and 3 support rotation and scaling.

Table 6.2 Backgrounds

Background	Max Resolution	Rotation/Scaling
0	512 x 512	No
1	512 x 512	No
2	128 to 1,024	Yes
3	128 to 1,024	Yes

Background Scrolling

The real advantage to tiled modes that I have yet to emphasize is the fact that these backgrounds can be layered on top of each other and that there is a priority involved in the layering, somewhat like a Z-buffer (if you lean toward the 3D realm). If you use video mode 0, with four text backgrounds (i.e., no scaling or rotation), then you can have four levels of parallax scrolling in your game, without any extra coding on your part (as far as writing the scrolling code or parallax layer transparency code, because that is all handled by the hardware). Most games that feature parallax are side scrollers, because it makes more sense to have scenery in the distance, with layers of terrain or objects closer to the player seeming to scroll by at a faster rate.

Mode 0 is great for this because all four backgrounds are hardware rendered. You do not need to write your own parallax scrolling routine. Now, you might be wondering, what is parallax scrolling? It's a concept that has been around for decades and is somewhat taken for granted today because it is so prevalent (kind of like a PC with ▇ 3D card, something that was once uncommon). Parallax scrolling involves multiple layers, with closer layers scrolling faster than the distant layers. Figure 6.3 shows a fictional game scene with ▇ starry background scrolling by slowly and a moonscape scrolling by more rapidly, with sprites transparently displayed on top of the two layers.

Tiles and Sprites

One of the primary advantages to using a tiled mode is that there is more memory available for hardware sprites in these modes, whereas in bitmap modes (3, 4, and 5) only half of the

VRAM is available for sprites. How you use that memory, based on sprite size, is up to you, and I will cover this subject in the next chapter. One thing is a given, though, that there are a maximum of 128 sprites available. If you want to write a scrolling shoot-'em-up, for instance, you would almost certainly want to use a tiled mode, if not for the hardware background scrolling, then certainly for the large number of supported sprites. Of course, you may use any combination of sizes for the sprites in your game. Although I am not covering sprites in this chapter, I hope this has piqued your interest, because sprites are covered in the next chapter, and it is a fun subject, building on the subjects covered in this chapter.

Figure 6.3
The foreground and background layers are scrolling at a different rate of speed. The spaceship and asteroids are sprites drawn over the backgrounds.

The Tile Data and Tile Map

The tile map is stored in the same location as the video buffer (in the bitmap video modes), an array of numbers that point to the tile images. In the text backgrounds (0 and 1) the tile map comprises 16-bit numbers, while the rotation backgrounds (2 and 3) store 8-bit numbers in the tile map. This is an important distinction that you should carefully remember because it can be a source of frustration when writing code, particularly when you switch to another background.

The GBA uses several registers to determine where the bitmaps are stored for the tiles displayed on the screen, which differs for each background. As you learned, the tile modes support two or more backgrounds each. The tile data itself can be stored anywhere in VRAM (video memory) as long as it is on a 16 KB boundary, which starts at 0x6000000 and goes through 0x600FFFF. When you are working with tile-based modes, video memory is divided

into four logical *character base blocks*, which are made up of 32 smaller *screen base blocks*, as shown in Figure 6.4.

The tile map (which defines where the tiles are positioned) must begin at screen base boundary 31 at the very end of video memory.

```
Char Base Block 0    Screen Base Block 0     0x6000000
                     Screen Base Block 1     0x6000800
                     Screen Base Block 2     0x6001000
                     Screen Base Block 3     0x6001800
                     Screen Base Block 4     0x6002000
                     Screen Base Block 5     0x6002800
                     Screen Base Block ▪     0x6003000
                     Screen Base Block 7     0x6003800

Char Base Block 1    Screen Base Block 8     0x6004000
                     Screen Base Block 9     0x6004800
                     Screen Base Block 10    0x6005000
                     Screen Base Block 11    0x6005800
                     Screen Base Block 12    0x6006000
                     Screen Base Block 13    0x6006800
                     Screen Base Block 14    0x6007000
                     Screen Base Block 15    0x6007800

Char Base Block 2    Screen Base Block 16    0x6008000
                     Screen Base Block 17    0x6008800
                     Screen Base Block 18    0x6009000
                     Screen Base Block 19    0x6009800
                     Screen Base Block 20    0x600A000
                     Screen Base Block 21    0x600A800
                     Screen Base Block 22    0x600B000
                     Screen Base Block 23    0x600B800

Char Base Block 3    Screen Base Block 24    0x600C000
                     Screen Base Block 25    0x600C800
                     Screen Base Block 26    0x600D000
                     Screen Base Block 27    0x600D800
                     Screen Base Block 28    0x600E000
                     Screen Base Block 29    0x600E800
                     Screen Base Block 30    0x600F000
                     Screen Base Block 31    0x600F800
```

Figure 6.4
Tile-based video memory is divided into logical blocks at 16 KB boundaries.

Creating a Scrolling Background

To demonstrate a tile-based scrolling background, I will walk you through a project called TileMode0, which you will write from scratch. Figure 6.5 shows the program running in the IDE.

For this program I cheated a little in order to make it easier to explain, at least for this first program in the chapter. What I mean by cheating is that I just created a single 256 x 256 bitmap image with all the tiles positioned already. In other words, this *does* use a tile map with tile images, but they are already set up, without the need for ▪ map editor or for manual placement. Figure 6.6 shows the bitmap image.
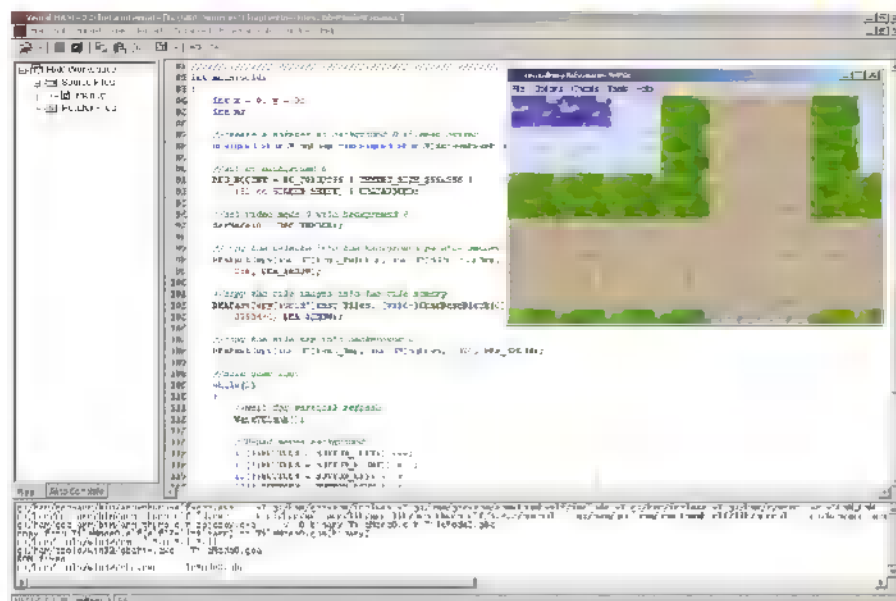
Figure 6.5
The TileMode0 program demonstrates a tiled scrolling background.
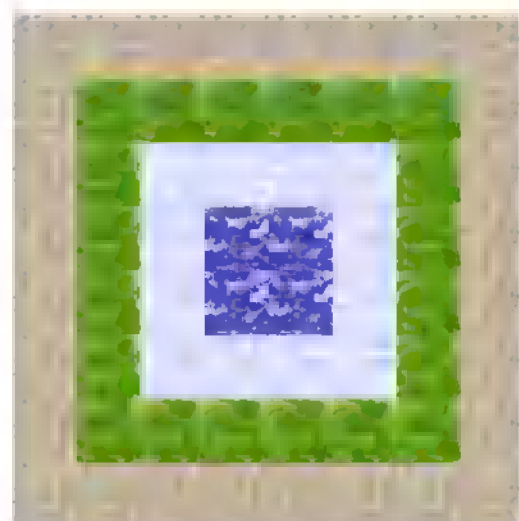
Figure 6.6
The bitmap file used as the source image for the tiles.

Now, there are only four different tiles in this image, so it's very wasteful to duplicate them throughout the image. The whole point of tiling is to create a single tile set and use it for the whole map. But like I said, this is a learning experience so I wanted to make it easier to understand. The next program, where I explain how to create a rotating background, will use ■ modified version of this tile set with just four tiles referenced in the map file (as shown in Figure 6.7).

Tile 0   Tile 1   Tile 2   Tile 3
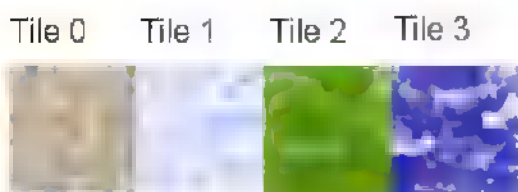
Figure 6.7
There are really only four tiles needed for the tile map.

Converting the Graphics

Before you can use the test.bmp file in a tile-based scroller, you'll need to convert it to a C array, just like you did with the sample programs in the previous chapter. This is very easy to do using the gfx2gba utility that is included with HAM. Before you can use it, you'll need to create a path to the program. Assuming you took my advice and installed HAM to the root under \HAM, then you can set up a HAM path by typing the following command into the command prompt (opened by selecting Start, Run, and typing "cmd" into the Run dialog box):

```
\ham\startham.bat
```

This batch file is included with the HAM SDK (which includes the GCC compiler, ARM assembler, and related tools). There is another option for starting a Command Prompt with support for the HAM tools, and that is a menu item in the Start menu that is provided by the HAM installer (that is, version 2.7 or later). Simply open Start, Program, HAM Development Kit, and click the option titled "HAM shell". That will open a Command Prompt that automatically runs startham.bat to set up the environment for running command-line tools.

Assuming you have copied the test.bmp file off the CD-ROM from \Sources\Chapter06\TileMode0 to your current project folder (where you plan to store the upcoming TileMode0 program), you can type this command:

```
gfx2gba -fsrc -m -ptest.pal -t8 test.bmp
```

Another, probably more convenient, method is to simply include the path to the utility when you run gfx2gba, like this (unless you used the "HAM shell" option, which I recommend):

```
\ham\tools\win32\gfx2gba -fsrc -m -ptest.pal -t8 test.bmp
```

The -m parameter tells the program to create a map file, while the -t8 parameter specifies a tile size of 8 x 8 pixels (the standard size supported by the GBA, which I wouldn't recommend changing, unless you are writing your own tile engine).

If gfx2gba was able to convert the file properly, you should see output that looks like this:

```
=====================

gfx2gba Converter v0.14

---------------------
```

```
Reading: test.bmp                (256x256 pixel, 256 colors)

Number of tiles before optimization: 1024

Number of tiles after  optimization: 0906

Saving tiled bitmap data to: test.raw.c ... ok

Saving map data to: test.map.c ... ok


Saving masterpalette to..: test.pal.c ... ok


Total files read & converted.:    1

Colors used before converting: 108

Colors used after converting.: 108

Colors saved.................:    0
```

If you pore over this output, you may notice something interesting, two lines that tell you how many tiles were created before and after optimization:

```
Number of tiles before optimization: 1024
Number of tiles after  optimization: 0906
```

Doing a little math, you can determine that there are 32 tiles across and 32 tiles down in this map, resulting in 1,024 total tiles (at 8 x 8 pixels each), based on the source 256 x 256 pixel image. But gfx2gba is a smart program and was able to optimize the tiles somewhat—not completely, or else it would have seen that there are a more limited number of tiles, but it does try to help. The test.bmp file that I created has four different "large" tiles, each of which is 32 x 32 pixels in size—meaning there are 16 of the 8 x 8 tiles in each one of my large tiles. At most, then, there should be only 64 tiles, rather than 906 tiles, but I don't particularly care because this is a first-time demo. I'll create an optimized tile map for the rotation program later.

Fast Blitting with DMA

One of the things that I have employed in this program to make it as fast as possible is a technique called DMA fast copy, which uses a special feature of the GBA to copy data from

one memory buffer to another—extremely fast. Basically, there's a custom chip on the GBA that handles memory—copying, moving, setting, clearing portions of memory, as well as normal accessing of data in memory by the CPU. Anytime the DMA chip is used, the CPU is temporarily suspended (only a matter of microseconds), until the DMA process is finished. This prevents the CPU from doing anything until a memory access is finished, otherwise problems could occur. Not only that, but in most computer systems and consoles, there is just one memory controller, and it can work on only one thing at a time. The newer memory architectures such as RDRAM and DDR found on PCs use two or more DMA controllers, meaning that memory can be accessed by two or more processes at the same time (or by the same process to access memory twice as fast). When the DMA chip is employed to write memory, it can't be used to read from memory at the same time, and vice versa. Therefore, the CPU is given a wait state while DMA activities are occurring.

DMA is a powerful aid to a GBA program, because it essentially replaces much source code with a single DMA call (or rather, three calls, as you will see shortly). Let's not forget also that DMA is a hardware process, where a software blitter is compiled and run by the CPU as machine instructions. You can't begin to compare a hardware process with a software process, because anything that is hard-coded into the silicon will blow away a series of machine instructions. For instance, the ARM7 CPU is a reduced instruction set computer (RISC) architecture, meaning that it has a small set of multipurpose instructions built in. More complex instructions must be built using what might be called building block instructions. Without getting into assembler language at this point (which is reserved for Chapter 12, "Optimizing Code with Assembly Language"), you could write a fast memory copy routine in assembler, and it would be much faster than a C routine. However, DMA will blow them both away when it comes to memory copies—and that is essentially what you need with a full-screen blitter. In fact, you don't even have to accommodate transparency in your code, because the GBA treats palette entry 0 as the transparent color. This could be useful for doing multilayer parallax backgrounds.

There are four DMA registers that you can use—or rather three, as the first one (# 0) is reserved. I'm just going to use the last register, although you could use DMA register 1, 2, or 3 just as well. Following is a list of the defines that you will need to use DMA fast copy for a background, as you will see in the upcoming TileMode0 program. The important defines here are the last two: DMA_32NOW and DMA_16NOW. These include options for copying 16-bit memory (such as external work RAM) and 32-bit memory (such as internal work RAM). For instance, the palette for a background is stored in a 16-bit memory address, while the tiles are stored in 32-bit memory.

```
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4

#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8

#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC

#define DMA_ENABLE 0x80000000

#define DMA_TIMING_IMMEDIATE 0x00000000

#define DMA_16 0x00000000

#define DMA_32 0x04000000

#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)

#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)
```

In order to perform a DMA fast copy, you simply set the three DMA registers to a value, and that triggers the process to start. Here is the DMAFastCopy function:

```c
void DMAFastCopy(void* source,void* dest,unsigned int count,unsigned int mode)
{
    if (mode == DMA_16NOW || mode == DMA_32NOW)
    {
        REG_DMA3SAD = (unsigned int)source;
        REG_DMA3DAD = (unsigned int)dest;
        REG_DMA3CNT = count | mode;
    }
}
```

Note how the function first makes sure that the two standard copy modes have been passed to it. Although there are other options that you could use, I am simply adding in this small level of error checking to keep the function from overwriting memory somewhere if an invalid option is passed to it. There are other time options, for instance, other than immediate. For instance, you can have DMA start the copy after a specified number of CPU clocks. I personally don't find utility in such features, because a fast copy should run immediately.

TileMode0 Source Code

This program has a lot of defines due to the various background mode and DMA settings used, but after you get past all the defines, the source code for the program is extremely short. It literally takes just a single line of code each to set up the palette, tiles, and map.

The most surprising thing about using backgrounds on the GBA is that you don't actually have to do any blitting code yourself; it is all done in the hardware, which is really bizarre if you are used to doing everything the hard way on a PC (for instance, using DirectX). On the GBA, once you have set the values into the appropriate locations in memory for the background settings, tile images, and tile map, the GBA handles the rest, including scrolling. In fact, to scroll the background, all you have to do is plug an X and Y value into the appropriate registers and—presto!—instant scrolling.

Now fire up Visual HAM and create a new project called TileMode0, or you may load this project off the CD-ROM from \Sources\Chapter06\TileMode0. You will need to have the test.map.c, test.pal.c, and test.raw.c files handy. If you skipped over the previous section on converting the tile graphics, you may want to go over that topic now, or simply copy the files off the CD-ROM. It's an invaluable lesson in creating tile maps, so I encourage you to go through process of converting the graphics rather than just using my premade files. Simply copy those files into the same folder where you created the new TileMode0 program, because this program includes those files. Since I spent so much time explaining how DMA works and how to initialize the background, and so on, I'm going to glaze over some of the other essentials for a scrolling background demo at this point and defer those explanations for the rotation example in the next section. Here is the source code for the TileMode0 program:

```
/////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 6: Tile-Based Video Modes
// TileMode0 Project
// main.c source code file
/////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


//include the sample tileset/map
#include "test.pal.c"
#include "test.raw.c"
#include "test.map.c"
```

```c
//function prototype
void DMAFastCopy(void*, void*, unsigned int, unsigned int);


//defines needed by DMAFastCopy
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC
#define DMA_ENABLE 0x80000000
#define DMA_TIMING_IMMEDIATE 0x00000000
#define DMA_16 0x00000000
#define DMA_32 0x04000000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)


//scrolling registers for background 0
#define REG_BG0HOFS *(volatile unsigned short*)0x4000010
#define REG_BG0VOFS *(volatile unsigned short*)0x4000012


//background setup registers and data
#define REG_BG0CNT *(volatile unsigned short*)0x4000008
#define REG_BG1CNT *(volatile unsigned short*)0x400000A
#define REG_BG2CNT *(volatile unsigned short*)0x400000C
#define REG_BG3CNT *(volatile unsigned short*)0x400000E
#define BG_COLOR256 0x80
#define CHAR_SHIFT 2
#define SCREEN_SHIFT 8
#define WRAPAROUND 0x1


//background tile bitmap sizes
#define TEXTBG_SIZE_256x256 0x0
#define TEXTBG_SIZE_256x512 0x8000
#define TEXTBG_SIZE_512x256 0x4000
#define TEXTBG_SIZE_512x512 0xC000
```

```c
//background memory offset macros
#define CharBaseBlock(n)  (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n)  (((n)*0x800)+0x6000000)


//background mode identifiers
#define BG0_ENABLE 0x100
#define BG1_ENABLE 0x200
#define BG2_ENABLE 0x400
#define BG3_ENABLE 0x800


//video identifiers
#define REG_DISPCNT *(unsigned int*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)
#define SetMode(mode) REG_DISPCNT = (mode)


//vertical refresh register
#define REG_DISPSTAT    *(volatile unsigned short*)0x4000004


//button identifiers
#define BUTTON_RIGHT 16
#define BUTTON_LEFT  32
#define BUTTON_UP     64
#define BUTTON_DOWN  128
#define BUTTONS (*(volatile unsigned int*)0x04000130)


//wait for vertical refresh
void WaitVBlank(void)
{
    while((REG_DISPSTAT & 1));
}


/////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
```

```
//////////////////////////////////////////////////////////////////
int main(void)
{
    int x = 0, y = 0;
    int n;

    //create a pointer to background 0 tilemap buffer
    unsigned short* bg0map =(unsigned short*)ScreenBaseBlock(31);

    //set up background 0
    REG_BG0CNT = BG_COLOR256 | TEXTBG_SIZE_256x256 |
        (31 << SCREEN_SHIFT) | WRAPAROUND;

    //set video mode 0 with background 0
    SetMode(0 | BG0_ENABLE);

    //copy the palette into the background palette memory
    DMAFastCopy((void*)test_Palette, (void*)BGPaletteMem,
        256, DMA_16NOW);

    //copy the tile images into the tile memory
    DMAFastCopy((void*)test_Tiles, (void*)CharBaseBlock(0),
        57984/4, DMA_32NOW);

    //copy the tile map into background 0
    DMAFastCopy((void*)test_Map, (void*)bg0map, 512, DMA_32NOW);

    //main game loop
    while(1)
    {
        //wait for vertical refresh
        WaitVBlank();

        //D-pad moves background
```

```
            if(!(BUTTONS & BUTTON_LEFT)) x--;
            if(!(BUTTONS & BUTTON_RIGHT)) x++;
            if(!(BUTTONS & BUTTON_UP)) y--;
            if(!(BUTTONS & BUTTON_DOWN)) y++;


            //use hardware background scrolling
            REG_BG0VOFS = y ;
            REG_BG0HOFS = x ;


            //wait for vertical refresh
            WaitVBlank();


            for(n = 0; n < 4000; n++);
    }
    return 0;
}


////////////////////////////////////////////////////////////
// Function: DMAFastCopy
// Fast memory copy function built into hardware
////////////////////////////////////////////////////////////
void DMAFastCopy(void* source, void* dest, unsigned int count,
    unsigned int mode)
{
    if (mode == DMA_16NOW || mode == DMA_32NOW)
    {
    REG_DMA3SAD = (unsigned int)source;
        REG_DMA3DAD = (unsigned int)dest;
        REG_DMA3CNT = count | mode;
    }
}
```

Creating a Rotating Background

Rotation backgrounds (2 and 3) are similar to text backgrounds in that they are made up of tiles in video modes 0, 1, or 2 (and behave differently in video modes 3, 4, or 5). But the so-called rotation backgrounds (obviously) support special features, such as rotation and scaling. I have written a sample program called RotateMode2 to demonstrate how to work with backgrounds 2 and 3. This program in particular uses background 2 and also runs in video mode 2—which, as you'll recall, supports the two rotation backgrounds, 2 and 3. For starters, let's create a new project in Visual HAM called RotateMode2. As usual, delete the default code that is inserted into main.c. I'll get into the source code as soon as I have finished explaining the tiles and map used in this program. The RotateMode2 program is shown in Figure 6.8.
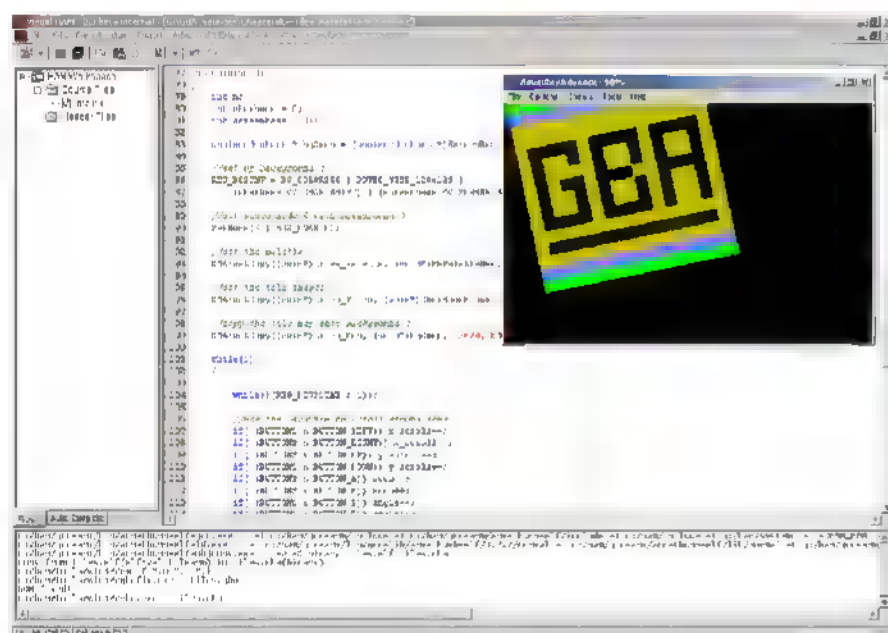
Figure 6.8

The RotateMode2 program demonstrates how to rotate a background.

Converting the Tile Image

This program uses a simple bitmap file to hold the five tiles used in the RotateMode2 program and is shown in Figure 6.9.

Figure 6.9

The simple tiles used in the RotateMode2 program.

To convert this program to a C array, like you did with the previous program in this chapter, you'll run gfx2gba with the following options:

```
\ham\tools\win32\gfx2gba -fsrc -m -t8 -rs -ptiles.pal tiles.bmp
```

These options specify a map file (-m), a tile size of 8 x 8 (-t8), and output for rotate/scale backgrounds (-rs).

Creating the Tile Map

Following is a listing of the tile map used in the RotateMode2 program. I scrapped the map generated by gfx2gba and created this one manually. First, this map is easier to read because it's not in hexadecimal, but rather it just shows simple decimal numbers. Second, this is a small map, so it's easy to see what the map looks like before actually running the program. You can also edit this map to see how your changes look when run. This map is stored in a file called tilemap.h and is included by the main program.

```
//16x16 tile map
const unsigned char tiles_Map[256] = {
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 3, 3, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 4, 4, 3, 4, 4, 4, 3, 3, 4, 4, 4, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

RotateMode2 Source Code

The source code for RotateMode2 follows. There are some new defines that you have not seen yet, notably the values and memory addresses needed by the rotation backgrounds, such as the following:

```
#define REG_BG2X        *(volatile unsigned int*)0x4000028

#define REG_BG2Y        *(volatile unsigned int*)0x400002C

#define REG_BG2PA       *(volatile unsigned short *)0x4000020

#define REG_BG2PB       *(volatile unsigned short *)0x4000022

#define REG_BG2PC       *(volatile unsigned short *)0x4000024

#define REG_BG2PD       *(volatile unsigned short *)0x4000026
```

which are used when rotating, scaling, and translating the background. The new rotation background defines are also needed:

```
#define ROTBG_SIZE_128x128    0x0

#define ROTBG_SIZE_256x256    0x4000

#define ROTBG_SIZE_512x512    0x8000

#define ROTBG_SIZE_1024x1024  0xC000
```

We'll be using the ROTBG_SIZE_128x128 define to set up the background. The key to this program, and to rotating backgrounds, is the RotateBackground function:

```
void RotateBackground(int ang, int cx, int cy, int zoom)
{
    center_y = (cy * zoom) >> 8;
    center_x = (cx * zoom) >> 8;


    DX = (x_scroll - center_y * SIN[ang] - center_x * COS[ang]);
    DY = (y_scroll - center_y * COS[ang] + center_x * SIN[ang]);


    PA = (COS[ang] * zoom) >> 8;
    PB = (SIN[ang] * zoom) >> 8;
    PC = (-SIN[ang] * zoom) >> 8;
    PD = (COS[ang] * zoom) >> 8;
}
```

Unfortunately, as you can see from this function, the GBA doesn't support hardware translation of the background in order to rotate it, as you must perform the rotation with your own code. The GBA does have registers set aside to actually do the pixel-by-pixel rotation, so at least that more difficult aspect is handled by the hardware.

The only prerequisite for this program is an external file called rotation.h, which must be in the same folder as the main program file. I won't list the file here because it's too long, and the listing is filled with long hexadecimal numbers. This file is necessary in order to perform the background rotation, as it contains the precalculated values for every one of the 360 degrees of rotation for both sine and cosine! Simply grab this file off the CD-ROM and put it in the RotateMode2 project folder. I have a different solution to sine and cosine in the next chapter that pre-calculates the sine and cosine at the start of the program, but this adds a short delay to the program's startup. For this reason, I leave you with these two solutions and let you decide which it better for your purposes.

Now let's get on with the full source code for this bad boy.

```
/////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 6: Tile-Based Video Modes
// RotateMode2 Project
// main.c source code file
/////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "rotation.h"
#include "tiles.pal.c"
#include "tiles.raw.c"
#include "tilemap.h"

//prototypes
void DMAFastCopy(void*, void*, unsigned int, unsigned int);
void RotateBackground(int, int, int, int);
```

```c
//defines needed by DMAFastCopy
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC
#define DMA_ENABLE 0x80000000
#define DMA_TIMING_IMMEDIATE 0x00000000
#define DMA_16 0x00000000
#define DMA_32 0x04000000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)


//background movement/rotation registers
#define REG_BG2X       *(volatile unsigned int*)0x4000028
#define REG_BG2Y       *(volatile unsigned int*)0x400002C
#define REG_BG2PA      *(volatile unsigned short *)0x4000020
#define REG_BG2PB      *(volatile unsigned short *)0x4000022
#define REG_BG2PC      *(volatile unsigned short *)0x4000024
#define REG_BG2PD      *(volatile unsigned short *)0x4000026


//background 2 stuff
#define REG_BG2CNT     *(volatile unsigned short *)0x400000C
#define BG2_ENABLE     0x400
#define BG_COLOR256    0x80


//background constants
#define ROTBG_SIZE_128x128     0x0
#define ROTBG_SIZE_256x256     0x4000
#define ROTBG_SIZE_512x512     0x8000
#define ROTBG_SIZE_1024x1024   0xC000
#define CHAR_SHIFT             2
#define SCREEN_SHIFT           8
#define WRAPAROUND             0x1
#define BG_MOSAIC_ENABLE       0x40
```

```
//video-related memory
#define REG_DISPCNT     *(volatile unsigned int*)0x4000000
#define BGPaletteMem    ((unsigned short *)0x5000000)
#define REG_DISPSTAT    *(volatile unsigned short *)0x4000004


#define BUTTON_A        1
#define BUTTON_B        2
#define BUTTON_RIGHT    16
#define BUTTON_LEFT     32
#define BUTTON_UP       64
#define BUTTON_DOWN     128
#define BUTTON_R        256
#define BUTTON_L        512
#define BUTTONS   (*(volatile unsigned int*)0x04000130)


#define CharBaseBlock(n)    (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n)  (((n)*0x800)+0x6000000)
#define SetMode(mode) REG_DISPCNT = (mode)


//some variables needed to rotate the background
int x_scroll=0,y_scroll=0;
int DX=0,DY=0;
int PA,PB,PC,PD;
int zoom = 2;
int angle = 0;
int center_y,center_x;


/////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////
int main(void)
{
```

```
int n;

int charbase = 0;

int screenbase = 31;


unsigned short * bg2map = (unsigned short *)ScreenBaseBlock(screenbase);


//set up background 0
REG_BG2CNT = BG_COLOR256 | ROTBG_SIZE_128x128 |

    (charbase << CHAR_SHIFT) | (screenbase << SCREEN_SHIFT);


//set video mode 0 with background 0
SetMode(2 | BG2_ENABLE);


//set the palette
DMAFastCopy((void*)tiles_Palette, (void*)BGPaletteMem, 256, DMA_16NOW);


//set the tile images
DMAFastCopy((void*)tiles_Tiles, (void*)CharBaseBlock(0), 256/4, DMA_32NOW);


//copy the tile map into background ▤
DMAFastCopy((void*)tiles_Map, (void*)bg2map, 256/4, DMA_32NOW);


while(1)

{

    while(!(REG_DISPSTAT & 1));


    //use the hardware to scroll around some
    if(!(BUTTONS & BUTTON_LEFT)) x_scroll--;

    if(!(BUTTONS & BUTTON_RIGHT)) x_scroll++;

    if(!(BUTTONS & BUTTON_UP)) y_scroll--;

    if(!(BUTTONS & BUTTON_DOWN)) y_scroll++;

    if(!(BUTTONS & BUTTON_A)) zoom--;

    if(!(BUTTONS & BUTTON_B)) zoom++;

    if(!(BUTTONS & BUTTON_L)) angle--;
```

```
            if(!(BUTTONS & BUTTON_R)) angle++;


            if(angle > 359)
                angle = 0;
            if(angle < 0)
                angle = 359;


            //rotate the background
            RotateBackground(angle,64,64,zoom);


            while((REG_DISPSTAT & 1));


            //update the background
            REG_BG2X = DX;
            REG_BG2Y = DY;
            REG_BG2PA = PA;
            REG_BG2PB = PB;
            REG_BG2PC = PC;
            REG_BG2PD = PD;


            while((REG_DISPSTAT & 1));
            for(n = 0; n < 100000; n++);


        }
    }


    ////////////////////////////////////////////////////////////////
    // Function: RotateBackground
    // Helper function to rotate a background
    ////////////////////////////////////////////////////////////////
    void RotateBackground(int ang, int cx, int cy, int zoom)
    {
        center_y = (cy * zoom) >> 8;
        center_x = (cx * zoom) >> 8;
```

```
        DX = (x_scroll - center_y * SIN[ang] - center_x * COS[ang]);

        DY = (y_scroll - center_y * COS[ang] + center_x * SIN[ang]);


        PA = (COS[ang] * zoom) >> 8;

        PB = (SIN[ang] * zoom) >> 8;

        PC = (-SIN[ang] * zoom) >> 8;

        PD = (COS[ang] * zoom) >> 8;

}


////////////////////////////////////////////////////////////

// Function: DMAFastCopy

// Fast memory copy function built into hardware

////////////////////////////////////////////////////////////

void DMAFastCopy(void* source, void* dest, unsigned int count,

    unsigned int mode)

{

    if (mode == DMA_16NOW || mode == DMA_32NOW)

    {

        REG_DMA3SAD = (unsigned int)source;

        REG_DMA3DAD = (unsigned int)dest;

        REG_DMA3CNT = count | mode;

    }

}
```

Summary

This has been one of the most challenging chapters of the book so far and was much more involved than the relatively simple video modes covered in the last chapter. However, now that you have conquered this difficult subject, you are on the downhill stretch of mastering the GBA, because you have now overcome the two most difficult subjects: bitmap and tile video modes. Get ready for even more graphics, as the next chapter finally covers the fascinating subject of sprite programming. Chapter 7 will involve even more of the subjects

covered in Chapters 5 and 6, giving you plenty of opportunity to practice using scrolling backgrounds and the like.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: The TileMode0 program used a 256 x 256 tile map and also tiled image. Modify the tiles and the source code, changing the tile map to 512 x 512, and note the differences in how fast the program runs.

Challenge 2: The RotateMode2 program uses a 128 x 128 tile map and corresponding tile image. However, the GBA supports rotation backgrounds of up to 1,024 x 1,024 in size! Modify the program to make use of this greater resolution.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. What are the three video modes 0, 1, and 2 called?
    A. Tiled backgrounds
    B. Bitmap backgrounds
    C. Cascading backgrounds
    D. Provocative backgrounds

2. How many backgrounds are available on the GBA, regardless of the video mode?
    A. 3
    B. 2
    C. 4
    D. 1

3. Which video mode features four tiled backgrounds?
    A. Mode 4
    B. Mode 0

C. Mode 7

D. Mode 2

4. Which backgrounds are supported by video mode 2?

    A. 1 and 2

    B. 1, 2, and 3

    C. 4, 5, and 6

    D. 2 and 3

5. Which video mode uses the two rotation backgrounds?

    A. Mode 0

    B. Mode 2

    C. Mode 3

    D. Mode 1

6. What three backgrounds are supported by video mode 1?

    A. 0, 1, and 2

    B. 1, 2, and 3

    C. 2 and 3

    D. 3, 4, and 5

7. Which of the three mode 1 backgrounds is considered a rotation background?

    A. 3

    B. 1

    C. 2

    D. 0

8. What are the following registers used for: REG_BG2PA, REG_BG2PB, REG_BG2PC, and REG_BG2PD?

    A. Background scrolling

    B. Background transparency

    C. Background hosiery

    D. Background rotation

9. How many registers are required to perform a DMA fast memory copy?

    A. 4

    B. 2

C. 3

D. 1

10. True/False: Does the GBA support hardware scrolling of the background?

A. True

B. False

Chapter 7

Rounding

Up Sprites

This chapter is an extension of the previous two chapters, which introduced the concept of bitmapped and tiled backgrounds and provided an overview of the six video modes available on the GBA. This chapter takes it a significant step further by building upon the concepts presented in those two chapters and adding additional material, most notably of which is coverage of sprites. Until now, all the graphics programming you have been doing has been directly on backgrounds in one or another of the six video modes. Some of the video modes were rather easy to draw upon, while others were significantly more difficult to get ■ pixel lit.

The point of this chapter is to refine that base knowledge and develop a sprite handler that incorporates all the code needed to deal with all the video modes and backgrounds built into the GBA, while at the same time providing significant coverage of the hardware sprite blitter. By the time you have finished this chapter, you will have a solid understanding of the most important aspect of Game Boy Advance programming: sprites.

Ready? Okay, let's go! This is the first "real-world" chapter that is more than a deluge of information—it actually gets into some of the fun factor involved in writing GBA programs. Until now, there was so much prerequisite information needed that it wasn't possible to write even ■ simple game—well, at least not a sprite-based game, which is the point after all. Here are the major topics presented in this chapter:

- Let's get serious: programming sprites
- Drawing a single sprite
- Creating a sprite handler
- Sprite special effects

Let's Get Serious: Programming Sprites

What is ▪ sprite, you ask? A sprite is a small, easily moved object that has a defined shape (usually withing ▪ rectangular image), and is the focus of the action in a 2D game (of which there is ▪ majority on the GBA). A sprite can be the player's space ship, role-playing character, baseball player, as well as a baseball, bird, missile, explosion, alien creature, or even ▪ vigilante's car. On most PC games, the game code itself must draw these sprites pixel by pixel; left to right, top to bottom. This is known as a software sprite. Console hardware, on the other hand, has traditionally provided hardware that can draw entire sprite bitmaps in ▪ single call or instruction. We call these hardware sprites.

Sprites have been a part of video games since the earliest days. Indeed, you can consider the ball and paddles of Pong to be sprites. Not all video game machines have had hardware support for sprites. The term *software sprite* has become common on systems like the PC where bitmaps reign supreme. The GBA has significant hardware support for sprites. In this chapter you will learn about object attribute memory (OAM) and examine a sprite handler to make sprites more manageable. Look at the definition of a sprite. We need to be able to specify a position and image for each of the sprites we want to display. The GBA also gives us a number of options that can be applied to the sprites. Setting these properties is the purpose of the OAM.

The GBA has built-in hardware support for up to 128 sprites. Each of the 128 sprites has the following attributes:

Tile Index. This specifies the image tile (or tiles) that holds the image of the sprite.

Size. Sprites can be several different sizes using from one to 64 "tiles" for the image. We'll talk more about tiles and how they relate to sprites in the next section.

Position. This specifies the horizontal and vertical position of the sprite on the screen.

Priority. This defines in which of four layers the sprite will drawn, allowing the sprite to show in front of or behind other graphics.

Palette Information. The tile graphics can use either 4 or 8 bits per pixel. One of 16 palettes must be chosen for 4 bit-per-pixel graphics.

Mosaic Effect. Sprites can have a mosaic effect applied.

Flip. Sprites can be flipped horizontally, vertically, or both.

Rotation and Scaling. Sprites can be rotated and scaled. Attributes specify which of 32 rotation and scaling parameters will be used.

These attributes are packed into 6 bytes of memory per sprite, but these structures are spaced 8 bytes apart. The extra 2 bytes of each chunk of memory are used for defining the sprite rotation and scaling parameters.

Moving Images the Simple Way

It is very easy to create moving objects on the screen using sprites, so let's look at each of these steps in detail.. The basic steps are as follows:

1. Create the sprite graphics.
2. Initialize the OAM.
3. Enable sprites in REG.DISPCNT.
4. Set the sprite attributes in OAM during VBlank for each sprite you want to display.
5. Any time an object moves or changes graphics, update the attributes in the OAM.

Creating Sprite Graphics

The image for a sprite is made up of one or more *tiles*. Each tile is an 8 x 8 bitmap of either 4 or 8 bits per pixel. The typical sprite is often larger than this and is seldom a solid rectangle.

Software sprites—moving objects drawn into a bitmap by software—handle the issue of transparency in a couple different ways.

One way to encode transparency is to have a *mask*—typically a 1 bit-per-pixel bitmap showing where there is solid sprite and where there is transparency. This is exactly like a crude alpha channel (where a color is made up of three channels (red, green, and blue) along with the alpha or transparency channel). One can then use this mask to erase what was in the bitmap and to combine the new graphics into the picture. This can be quite expensive in software.

Another way to encode transparency is to have a specific color that represents transparent pixels. This keeps one from wasting space on a separate mask bitmap but still requires a comparison for every pixel—again very expensive.

The GBA sprite hardware uses a variant of this second method. Since sprites always use color palettes (of either 16 or 256 colors), color index zero is set aside for transparency. This is true for every graphics mode on the GBA that uses palettes—the first color entry in the palette specifies the transparent pixels of the image. Therefore, when you are creating game graphics in a graphic editor, be sure to modify the palette so that the transparent color is in the first position.

Creating Tiles

The easiest way to create the data for tiles is to draw the images in a larger bitmap and then use a utility program to chop the bitmap up into the tile data. We'll use the same graphics converter we used before, gfx2gba for this purpose, along with another tool called pcx2sprite that is particularly suitable for single sprites. There is an ideal width to use for

creating sprite tiles. This width is different depending on the color depth you are using. We'll see why this is important in a little bit.

For This Color Depth	Use This Width
16 colors	256 pixels
256 colors	128 pixels

I find it easiest to work on these graphics by zooming in on them. A zoom factor of 4 x to 8 x works really well for me. Turn on the Grid option and set the grid to 8 pixels for width and height. Each square of the grid shows the boundaries of one tile. Using the ideal bitmap widths you will have 32 tiles per row for 16-color tiles or 16 tiles per row for 256-color tiles

Converting Tiles

The pcx2sprite program has no parameters, and it is convenient because you can just drag a .pcx file over the program file in Windows Explorer to convert the file (note that only 256 colors are supported). The other tool that is still needed for backgrounds is gfx2gba. The parameters most commonly used with gfx2gba are as follows:

-t8	Sets the size of a tile to 8 x 8 pixels.
-c32k	Use hicolor for mode 3 (not needed for tiled modes). The default is 256 even if the source files are 16 color files.
-pFilename	Sets the name for the palette file.
-fsrc	Output will be in source code format.

> *Bitmap graphics modes 3, 4, and 5 use the first half of the Sprite Tile VRAM for part of their buffers. This limits the sprite tiles to the last 16 KB of VRAM—512 16-color tiles or 256 256-color tiles. In both of these cases the first usable sprite tile is index 512.*

Using Sprites

The mechanics of using sprites are fairly simple but can cause some annoying problems. Some examples: The data stored in the OAM is packed with various single and multibit quantities. The OAM needs to be updated during the vertical refresh period. Sprites of the same display priority are sorted by OAM position (lowest sprite number has priority).

Because of these issues, most games use a separate buffer typically known as *shadow memory*. The shadow has the identical bit layout as the OAM but is located in RAM where it can be modified without affecting the display. The sprite attribute array is then copied into

the OAM at the start of the vertical refresh. Note that you don't need to bother with the sprite images after having initially copied them to the data portion of OAM..

Larger Sprites

Let's face it. Sprites that are 8 x 8 pixels just aren't very large. To make recognizable characters and animations, you'll want to use multiple tiles per sprite. Here's where the natural size of the source bitmaps comes into play. As noted before, the natural size is different for 16- and 256-color sprites. That's because the 256-color tiles use twice as much memory as the 16-color tiles.

The 256-color tiles each take 64 bytes of memory. The first 256-color tile uses the memory from 0x06010000 to 0x0601003f. This is the same memory that tiles 0 and 1 take up in 16-color tiles. The second 256-color tile starts at 0x06010040—the same address as tile index 2 of 16-color tiles. Any guesses what index value we use for this second 256-color tile? Right, index 2. The formula for converting a tile index to memory addresses remains the same in the two modes, but the 256-color tiles only use the even indexes. This also means that there are only 512 tiles in this mode.

Linear Tile Layouts

The tile arrangements that you've seen so far are the default and are known as the 2D tile arrangement. This layout is very convenient for the tile artists since they can simply draw the larger sprites in a bitmap and the conversion tools directly give us usable sprite orderings. For many games this is fine because 1,024 (or 512) tiles are often enough for all the sprites in one level of a game.

There are many times when this is not the case. Games with a lot of large characters or long animation sequences will not be able to fit all their graphics for a level in the 32 KB provided for the sprite tiles. This means you need to dynamically load graphics data from your game ROM (or EWRAM) during gameplay.

Dealing with the 2D tile layout while dynamically loading sprite tiles is inefficient and likely to cause severe brain damage to the programmer.

You can flip one bit in REG.DISPCNT and change the layout of tiles for all sprites.

```
REG.DISPCNT |= DC_SPRITESEQ;
```

This bit sets the sprite's tiles to a sequential layout. This means that instead of adding 32 to get the index of the first tile on the next row, this tile immediately follows the last tile on the previous row. This mode keeps all the graphics data for a single sprite image in contiguous memory allowing a single DMA transfer to move an entire image. There are tools that will convert ■ bitmaps into tiles arranged in this manner. The gfx2gba utility will do this for you using its "tiling" option. Another tool that comes with the HAM SDK, called the "Bitmap ReSizer" (see Start, Programs, HAM Development Kit, Tools, Bitmap ReOrganizer),

that will convert a tiled bitmap into a linear format, and is a windowed program, rather than a command-line program. Of course, this program works only with bitmaps, and does not convert them. You still need to use gfx2gba to convert the image.

Drawing a Single Sprite

Now that you've had a little theory, how about delving into some real code for drawing sprites on the screen? The first sample program in this chapter is called SimpleSprite and is shown in Figure 7.1. As you can see from the figure, the SimpleSprite program draws a spaceship sprite and moves it across the screen, warping to the left when it hits the right side of the screen.

This program has just enough code to get you going, without a lot of complicated extraneous stuff because I want you to first grasp how to convert a sprite file and then display it on the screen. In later sections, I'll get into special effects like rotation, scaling, and transparency, as well as how to handle multiple sprites. In fact, you will be writing a sprite handler before this chapter is over.

Converting the Sprite

The first thing you need in a sprite-based program is an image of a sprite, which can be anything: a spaceship, car, soldier, lemming, ghost, hero, monster. Basically, this is the key to the game, your graphics! Figure 7.2 shows an image of a spaceship stored in ship.pcx. You can find this file in \Sources\Chapter07\SimpleSprite, along with the source files for this project.
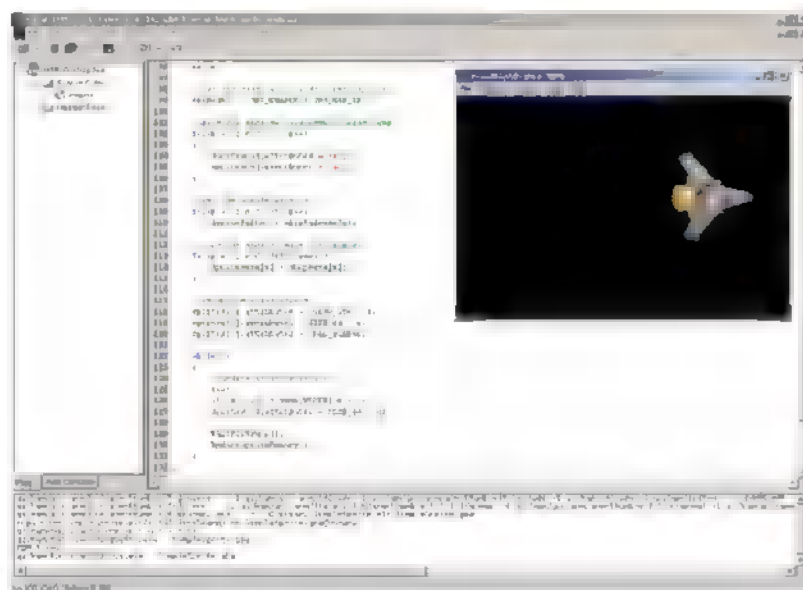
Figure 7.1

The SimpleSprite program moves a single sprite across the screen.

Figure 7.2

*The ship.pcx file used as the sprite
in the SimpleSprite program.*

Also included in the folder for this project is a program called pcx2sprite.exe. This program is really handy for quick conversion of sprites because it doesn't require a command-line interface. You simply drag a .pcx file over the program file in Windows Explorer, and it converts it to a C source code file—which is similar to the files produced by gfx2gba, but pcx2sprite puts the palette and bitmap inside the same file.

> *You can download pcx2sprite, pcx2gba, and many other utilities, from the Pern Project Web site at http://www.thepernproject.com, operated by Jason Rogers (a.k.a. Dovoto).*

Open Windows Explorer, browse to \Sources\Chapter07\SimpleSprite, and locate ship.pcx. Now drag this file over the pcx2sprite.exe file to convert it. The file must be a 256-color .pcx image, otherwise pcx2sprite will output an error message. The output is shown in Figure 7.3.
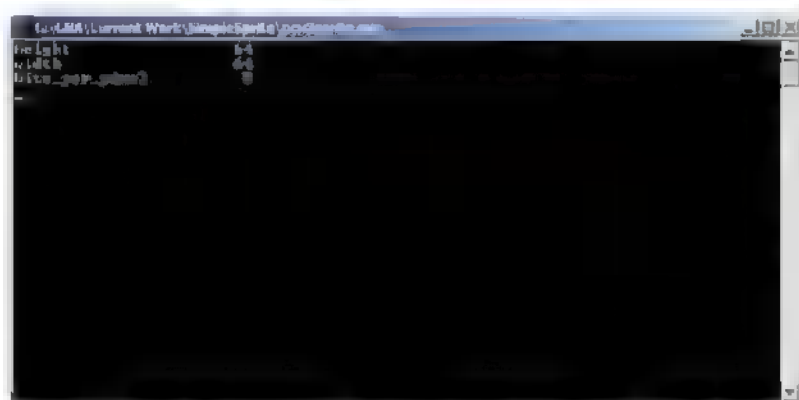
Figure 7.3

*Output from the
pcx2sprite program.*

After that is done, you'll have a file called ship.h. If you open the file, you'll see something like this:

```
/*********************************************\
*        ship.h                              *
*           by dovotos pcx->gba program      *
/*********************************************/
#define  ship_WIDTH    64
```

```
#define  ship_HEIGHT   64


const u16 shipData[] = {

0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
0x0000,  0x0000,  0x0000,  0x1301,  0x0000,  0x0000,  0x0000,  0xFB50,  0x0000,  0x0000,
0x1900,  0xE9AD,  0x0000,  0x0000,

...
```

Only the top few lines are shown from a file that is a few hundred lines long, but this gives you an idea of what the ship.h file looks like after conversion. Since the compiler complains about Dovoto's funky header at the top, I always just delete the header.

The SimpleSprite Source Code

The SimpleSprite program has just a single source listing with all the defines and stuff you need to compile the program in one place. Create a new project in Visual HAM, name it SimpleSprite, and delete the default code in main.c, replaced with the following code listing:

```
/////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// SimpleSprite Project
// main.c source code file
/////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


typedef unsigned short u16;


#include "ship.h"


//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```
//define some video addresses

#define REG_DISPCNT *(volatile unsigned short*)0x4000000

#define BGPaletteMem ((unsigned short*)0x5000000)

#define REG_VCOUNT *(volatile unsigned short*)0x4000006

#define REG_DISPSTAT *(volatile unsigned short *)0x4000004


//define object attribute memory state address

#define SpriteMem ((unsigned short*)0x7000000)


//define object attribute memory image address

#define SpriteData ((unsigned short*)0x6010000)


//define object attribute memory palette address

#define SpritePal ((unsigned short*)0x5000200)


//misc sprite constants

#define OBJ_MAP_2D          0x0

#define OBJ_MAP_1D          0x40

#define OBJ_ENABLE          0x1000


//attribute0 stuff

#define ROTATION_FLAG       0x100

#define SIZE_DOUBLE         0x200

#define MODE_NORMAL         0x0

#define MODE_TRANSPARENT    0x400

#define MODE_WINDOWED       0x800

#define MOSAIC              0x1000

#define COLOR_16            0x0000

#define COLOR_256           0x2000

#define SQUARE              0x0

#define TALL                0x4000

#define WIDE                0x8000


//attribute1 stuff
```

```
#define ROTDATA(n)              ((n) << 9)

#define HORIZONTAL_FLIP         0x1000

#define VERTICAL_FLIP           0x2000

#define SIZE_8                  0x0

#define SIZE_16                 0x4000

#define SIZE_32                 0x8000

#define SIZE_64                 0xC000


//attribute2 stuff
#define PRIORITY(n)             ((n) << 10)

#define PALETTE(n)              ((n) << 12)


//sprite structs
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;


//create an array of 128 sprites equal to OAM
Sprite sprites[128];


//function prototypes
void WaitForVsync(void);
void UpdateSpriteMemory(void);


/////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////
int main(void) {
    signed short x = 10, y = 40;
```

```c
signed short xdir = 1, ydir = 1;

int char_number = 0;

int n;


//set the video mode--mode 2 with sprites
   SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);


//move all sprites offscreen to hide them
for(n = 0; n < 128; n++)
{
    sprites[n].attribute0 = 160;
    sprites[n].attribute1 = 240;
}


//set the sprite palette
for(n = 0; n < 256; n++)
    SpritePal[n] = shipPalette[n];


//copy the sprite image into memory
for(n = 0; n < 256*8; n++) {
    SpriteData[n] = shipData[n];
}


//setup the first sprite
sprites[0].attribute0 = COLOR_256 | y;
sprites[0].attribute1 = SIZE_64 | x;
sprites[0].attribute2 = char_number;


while(1)
{
    //update sprite x position
    x += xdir;
    if (x > 239 - ship_WIDTH) x = 0;
```

```
            //update sprite y position
            y += ydir;
            if (y > 159 - ship_HEIGHT)
            {
                y = 159 - ship_HEIGHT;
                ydir = -1;
            }
            if (y < 1)
            {
                y = 1;
                ydir = 1;
            }


            //update sprite attributes with new x,y position
        sprites[0].attribute0 = COLOR_256 | y;
            sprites[0].attribute1 = SIZE_64 | x;


            //wait for vertical retrace
            WaitForVsync();


            //display the sprite
            UpdateSpriteMemory();
        }
    }


//////////////////////////////////////////////////////////////
// Function: WaitForVsync
// Waits for the vertical retrace
//////////////////////////////////////////////////////////////
void WaitForVsync(void)
{
    while((REG_DISPSTAT & 1));
}
```

```
/////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
/////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128*4; n++)
        SpriteMem[n] = temp[n];
}
```

I am basically going to follow a learn-by-doing philosophy in this chapter (and others) because a brief glance at a snippet of code often explains more than ■ dozen pages of commentary.

Creating a Sprite Handler

Now that you've seen what might be called crude sprite code running, I'd like to show you ■ few tricks that will make sprite handling more manageable. Writing a game entirely with GBA sprite code is possible, and there's nothing wrong with doing it that way. Many have done that without any trouble. But I prefer to keep track of sprites with a handler, which is basically a struct with basic sprite values stored inside, such as x, y, size, xdir, ydir, and so on. Over the next few sections I'll improve the basic sprite handler so it incorporates more features in time (such as rotation, scaling, and transparency).

What Does the Sprite Handler Do?

The sprite handler basically keeps track of the sprites in the program so you don't have too many global variables floating around. Here's what the struct looks like at first revision:

```
typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
```

```
    int size;
}SpriteHandler;
```

Actually using the struct involves creating an array of structs:

```
SpriteHandler mysprites[128];
```

Now, with this code in place, you have much more control over the sprites in your program, and you can manage them in large quantity without an exponential increase in the amount of code. In fact, you can simply process all the sprites in a for loop.

The BounceSprite Source Code

Let's put the sprite handler to use, shall we? This section includes a program called BounceSprite, which draws a background image and displays several sprites on the screen, bouncing them around inside the dimensions of the screen. The output is shown in Figure 7.4.
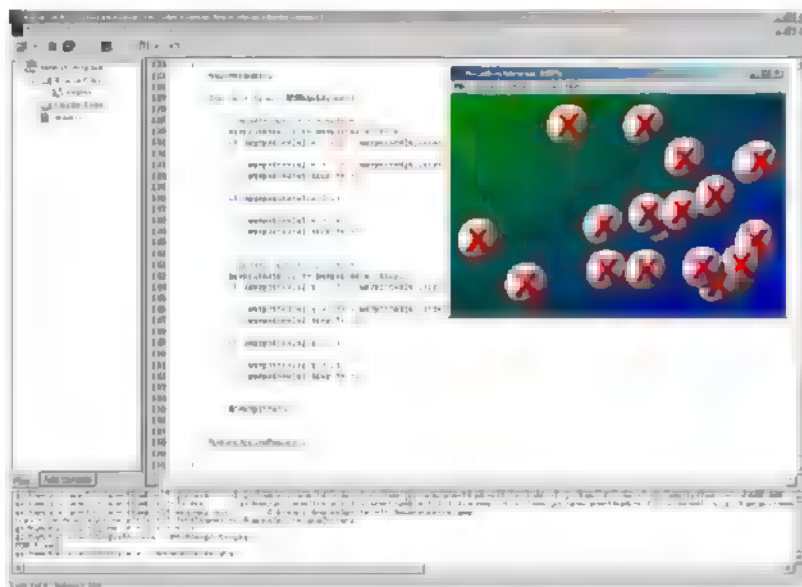
Figure 7.4

The BounceSprite program bounces 10 ball sprites around on the screen.

I have divided this program into two separate source files to make it easier to follow. While the SimpleSprite program was somewhat short, the BounceSprite program is a little more involved because of the new handler code. It also does quite a bit more than the SimpleSprite program, as there are now 10 sprites on the screen, moving independently.

Go ahead and create a new project in Visual HAM called BounceSprite, or you may copy the project off the CD-ROM, located in \Sources\Chapter07\BounceSprite. If you are typing in the program, you'll want to add a new file to the project. Select File, New, New File to bring up the New File dialog box, as shown in Figure 7.5.
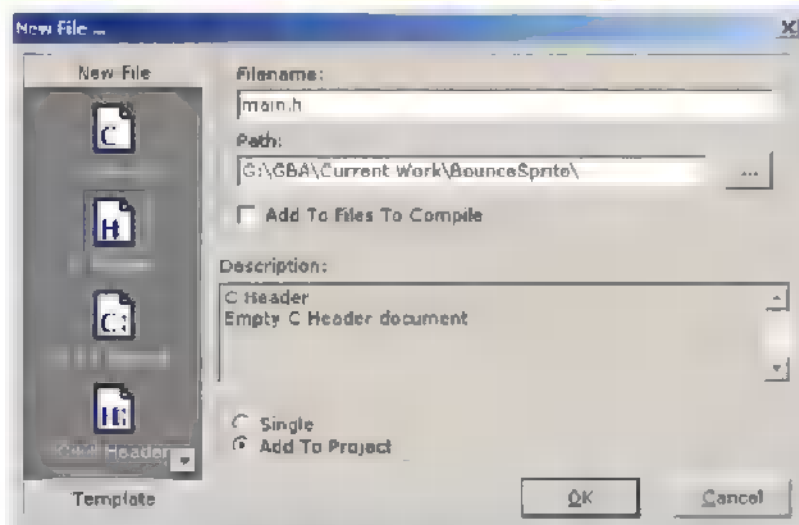
Figure 7.5

The New File dialog box is used to add new source files to the project.

Select the C Header file type at the left, then type "main.h" for the new file name, and be sure to select the option Add To Project before closing the dialog box. The new main.h file should now be in your BounceSprite project.

The Header File

Type the following code into the main.h file:

```
//////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// BounceSprite Project
// main.h header file
//////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "ball.h"
#include "bg.raw.c"

//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```
//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)


//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;


//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)


//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)


//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;


//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)


//misc sprite constants
#define OBJ_MAP_2D 0x0
#define OBJ_MAP_1D 0x40
#define OBJ_ENABLE 0x1000
#define BG2_ENABLE0x400


//attribute0 stuff
#define ROTATION_FLAG       0x100
#define SIZE_DOUBLE         0x200
#define MODE_NORMAL         0x0
```

```c
#define MODE_TRANSPARENT    0x400
#define MODE_WINDOWED       0x800
#define MOSAIC              0x1000
#define COLOR_256           0x2000
#define SQUARE              0x0
#define TALL                0x4000
#define WIDE                0x8000


//attribute1 stuff
#define SIZE_8              0x0
#define SIZE_16             0x4000
#define SIZE_32             0x8000
#define SIZE_64             0xC000


//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;


//create an array of 128 sprites equal to OAM
Sprite sprites[128];


typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;


}SpriteHandler;
```

```
SpriteHandler mysprites[128];


#endif
```

The Main Source File

The main source code file for BounceSprite is listed next. Since most of the building blocks are now stored in main.h, this code listing is much more manageable than it would have otherwise been. Note the **#define NUMBALLS 10** definition. You may change that to another number if you wish, to see how the program performs with differing numbers of sprites. Take care, however, because this program is using large sprites, so there are not a full **128** slots available in memory.

```
//////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// BounceSprite Project
// main.c source code file
//////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


#include "main.h"


#define NUMBALLS 10


//////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
```

```
            sprites[n].attribute0 = 160;

            sprites[n].attribute1 = 240;

        }

    }


    //////////////////////////////////////////////////////////

    // Function: MoveSprite

    // Changes sprite attributes for x,y positions

    //////////////////////////////////////////////////////////

    void MoveSprite(int num)

    {

        //clear the old x value

        sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;

        sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;


        //clear the old y value

        sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;

        sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;

    }


    //////////////////////////////////////////////////////////

    // Function: UpdateSpriteMemory

    // Copies the sprite array into OAM memory

    //////////////////////////////////////////////////////////

    void UpdateSpriteMemory(void)

    {

        int n;

        unsigned short* temp;

        temp = (unsigned short*)sprites;


        for(n = 0; n < 128 * 4; n++)

            SpriteMem[n] = temp[n];

    }
```

```c
/////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
/////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

    sprites[num].attribute1 = sprite_size | x;
}


/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
```

```c
void WaitVBlank(void)
{
    while(*ScanlineCounter < 160);
}


///////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
///////////////////////////////////////////////////////
int main()
{
    int n;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //load ball sprite
    for(n = 0; n < 512; n++)
        SpriteData3[n] = ballData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image (512)
    for (n = 0; n < NUMBALLS; n++)
    {
```

```
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
            COLOR_256, 512);

    while (mysprites[n].dirx == 0)
        mysprites[n].dirx = rand() % 6 - 3;

    while (mysprites[n].diry == 0)
        mysprites[n].diry = rand() % 6 - 3;

}


//main loop

while(1)

{

    //keep the screen civil
    WaitVBlank();

    for (n = 0; n < NUMBALLS; n++)
    {
        //update sprite x position
        mysprites[n].x += mysprites[n].dirx;
        if (mysprites[n].x > 239 - mysprites[n].size)
        {
            mysprites[n].x = 239 - mysprites[n].size;
            mysprites[n].dirx *= -1;
        }
        if (mysprites[n].x < 1)
        {
            mysprites[n].x = 1;
            mysprites[n].dirx *= -1;
        }


        //update sprite y position
        mysprites[n].y += mysprites[n].diry;
        if (mysprites[n].y > 159 - mysprites[n].size)
        {
            mysprites[n].y = 159 - mysprites[n].size;
```

```
                mysprites[n].diry *= -1;

        }

        if (mysprites[n].y < 1)

        {

                mysprites[n].y = 1;

                mysprites[n].diry *= -1;

        }


        //update the sprite properties

        MoveSprite(n);

    }

    //copy all sprites into object attribute memory

    UpdateSpriteMemory();

    }

}
```

Resizing the Ball Sprite

This sprite handler is not fully featured, but it does allow you to change the sprite size on the fly without modifying the source code in any way. The reason this is possible is because the pcx2sprite program supplies the image width and height in the converted source file (such as ball.h). The InitSprite function has a parameter that specifies the dimensions of the sprite. For all practical purposes, you will be using square sprites, with the same width and height, so it makes sense to use a single parameter, size, for the dimensions. If your particular situation calls for rectangular sprite images, then you may modify the program to use a width and height. This is but a stepping stone on the way to delivering sprites to the screen, however.

As Figure 7.6 shows, we're on the right track, but these GBA sprites are capable of animation, not to mention special effects like scaling and rotation. This minor change is also ▪ demonstration of sprite performance, as the BounceSprite2 program increases the NUMBALLS to 128 but is otherwise identical to the BounceSprite program. There is an additional need to grab frames out of an image to use for individual sprites, so what we need is a function for loading sprites, in the traditional sense. Let's get into some special effects now, and I'll cover tiled sprite images at the same time.
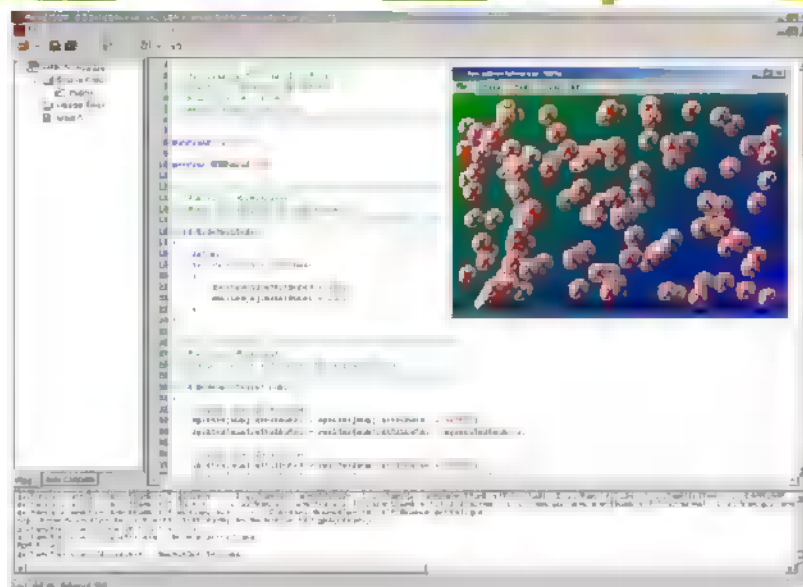
Figure 7.6

Resizing the ball image to 16 x 16 requires absolutely no change in the source code, because image dimensions are stored in the converted file by pcx2sprite.

Sprite Special Effects

The GBA is ■ highly optimized sprite-blitting machine and provides some extra features in addition to transparency—which, in case you didn't notice, was in use with the BounceSprite program, and this was all done automatically by the GBA hardware. The traditional use of transparency is to show only solid pixels in the sprite, thus allowing transparent pixels to show what is behind the sprite (on the background, for instance). The GBA takes care of this for you—something that requires quite a heaping of theory and code in other platforms! There is another form of transparency—or rather, translucency—and that is called *alpha blending*.

Implementing Alpha Blending

Alpha blending is a technique whereby one image is translucent, allowing the images behind it to show through, while still remaining visible itself. The effect is extremely useful not only for sprite special effects but also for displaying dialogs or other images over the game screen while still showing the game in the background. One such example is an options screen that might appear when you press the Start button, providing options such as restart, save, load, and quit. Displaying a menu in a translucent dialog has a very nice effect on the screen, with the appearance of being less invasive.

There are really only a few things that you must do to enable alpha blending of foreground sprites. First, you'll need two new registers, **REG_BLDMOD** and **REG_COLEV**:

```
//transparency registers

#define REG_BLDMOD     *(unsigned short*)0x4000050
```

```
#define REG_COLEV        *(unsigned short*)0x4000052
```

As usual, these are pointers to memory addresses where the hardware defines these features. Put into use, translucency (alpha blending) can be turned on with the following code:

```
//set transparency level
REG_BLDMOD = (1 << 4) | (1 << 10);
REG_COLEV = (8) + (8 << 8);
```

When you set these two registers as shown, any sprites that have transparency enabled will appear so, while those without the option will be displayed normally. Here are the two definitions of the options used to set up a sprite for transparency:

```
#define MODE_NORMAL         0x0
#define MODE_TRANSPARENT    0x400
```

Where do you use these definitions? The object attribute memory (OAM) sprite struct has four attributes that are used to specify various options for each sprite. One such attribute is attribute0, which takes care of the color mode, rotation factor, the vertical (y) position, as well as the transparency of the sprite. It is unfortunate that so much is crammed into each attribute; perhaps you will figure out a way to rewrite the struct with individual attributes for each setting? It would require a lot of tinkering to figure out all the bits but may be worth the attempt. I will stick with the standard way of modifying sprites. I realize it is confusing that *transparent* is used to mean alpha blending, as well as a transparent sprite color. But I think we can get away with the terminology when dealing with the GBA because the traditional use of *transparency* is handled by the GBA hardware already, so you really aren't going to be dealing with that aspect at all (I was tempted to say "very often," but really, the GBA does this entirely for you). Here is how you would set up attribute0 to enable transparency:

```
sprites[num].attribute0 = COLOR_256 | MODE_TRANSPARENT | y;
```

Blitting Transparent Sprites

I've written a sample program called TransSprite, which I'd like to walk you through. It's a short program, like all the other sample programs in this chapter, so it takes but a few minutes to type it in. As usual, create a new project in Visual HAM. Name the new project TransSprite. The program is located on the CD-ROM under \Sources\Chapter07\TransSprite. Figure 7.7 shows the TransSprite program running. In this particular screen shot, the sprites are all transparent (or rather, translucent, or alpha blended).

If you watch the TransSprite program run for a few seconds, you'll see the sprites alternate from solid to transparent. Figure 7.8 shows the two variations of the program side by side.

In this case, the VisualBoyAdvance screen is shown actual size (whereas I normally run it at 2 x).

Well, it seems as if you have enough to go on already to implement alpha blending of sprites. How about a sample program, just to put into good use what you have learned? It's great being able to get instant feedback on some new technology—one of the joys of programming (and the reason why there is a field called *computer science*).
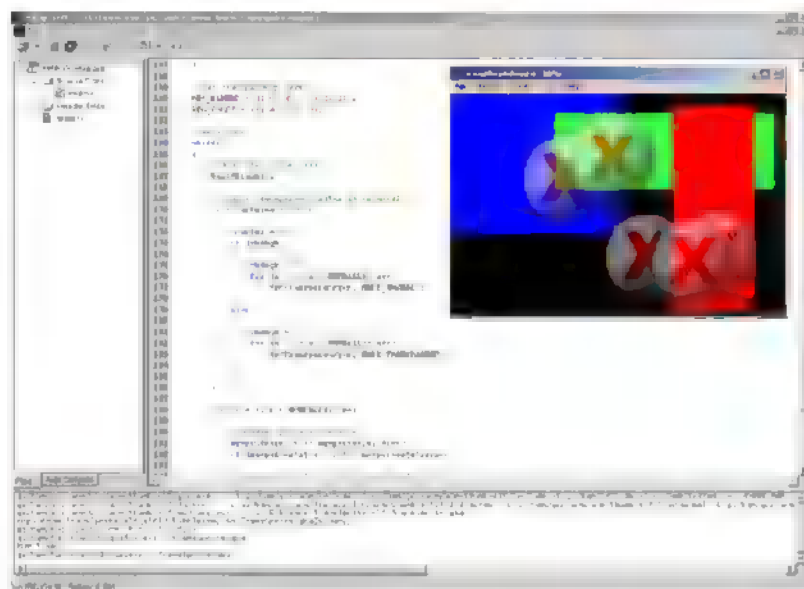
Figure 7.7

The TransSprite program demonstrates how to turn on alpha blending, which allows sprites to be drawn transparently.
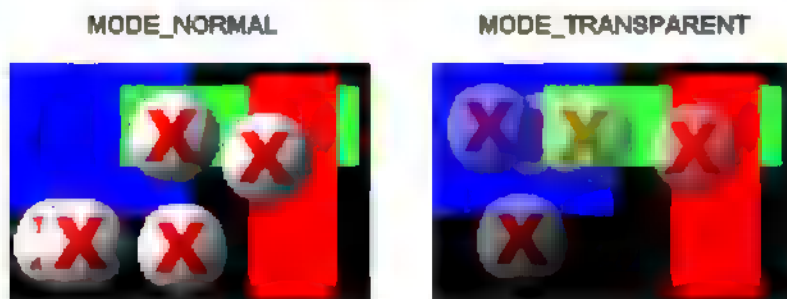
Figure 7.8

The TransSprite program alternates the sprites from MODE_NORMAL to MODE_TRANSPARENT.

Now let's create a project for this program. In Visual HAM, open the File menu and select New, New Project. Name the project TransSprite. As you did earlier with the previous project, add a new header file called main.h.

The TransSprite Header File

The header file for the TransSprite program is called main.h and contains all the includes, defines, arrays, variables, and GBA registers needed by the main program and is extremely welcome because none of these statements ever change while the program is running, so it's

better to hide them away. As long as you know what all of these statements are for, and how they were derived, that's the whole point of the lesson. So let's hide them away in main.h.

```c
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// TransSprite Project
// main.h header file
//////////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "ball.h"
#include "bg.raw.c"

//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)
```

```
//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)


//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;


//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)


//transparency stuff
#define REG_BLDMOD        *(unsigned short*)0x4000050
#define REG_COLEV         *(unsigned short*)0x4000052


//misc sprite constants
#define OBJ_MAP_2D            0x0
#define OBJ_MAP_1D            0x40
#define OBJ_ENABLE           0x1000
#define BG2_ENABLE           0x400


//attribute0 stuff
#define ROTATION_FLAG        0x100
#define SIZE_DOUBLE          0x200
#define MODE_NORMAL          0x0
#define MODE_TRANSPARENT     0x400
#define MODE_WINDOWED        0x800
#define MOSAIC               0x1000
#define COLOR_256            0x2000
#define SQUARE               0x0
#define TALL                 0x4000
#define WIDE                 0x8000


//attribute1 stuff
#define SIZE_8               0x0
```

```
#define SIZE_16              0x4000
#define SIZE_32              0x8000
#define SIZE_64              0xC000


//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;


//create an array of 128 sprites equal to OAM
Sprite sprites[128];


typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;
    int colormode;
    int trans;


}SpriteHandler;


SpriteHandler mysprites[128];


#endif
```

Did you notice the two new elements in the SpriteHandler struct that I snuck in? The new elements are colormode and trans and are provided to allow each sprite to have separate and distinct properties from all others. You'll be adding more items to the struct in later projects as well, so don't get too comfortable with the sprite handler just yet.

The TransSprite Source File

The main source code file for TransSprite may be the most lengthy code listing of the chapter so far, but it is not inefficient by any means. Given what this program does, it is quite small compared to the amount of code needed to implement alpha blending on another platform. Now here is the code listing for the main source code file:

```c
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// TransSprite Project
// main.c source code file
//////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

#define NUMBALLS 5

//////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

//////////////////////////////////////////////////////////
```

```
// Function: MoveSprite

// Changes sprite attributes for x,y positions

//////////////////////////////////////////////////////

void MoveSprite(int num)

{

    //clear the old x value

    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;

    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;


    //clear the old y value

    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;

    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;

}



//////////////////////////////////////////////////////

// Function: UpdateSpriteMemory

// Copies the sprite array into OAM memory

//////////////////////////////////////////////////////

void UpdateSpriteMemory(void)

{

    int n;

    unsigned short* temp;

    temp = (unsigned short*)sprites;


    for(n = 0; n < 128 * 4; n++)

        SpriteMem[n] = temp[n];

}
```

The InitSprite () function that follows is where the sprite attribute is modified to enable alpha blending of the sprite. I have highlighted the key line in bold text.

```
//////////////////////////////////////////////////////

// Function: InitSprite

// Initializes a sprite within the sprite handler array

//////////////////////////////////////////////////////

void InitSprite(int num, int x, int y, int size, int tileIndex)
```

```
{
    unsigned int sprite_size = 0;


    mysprites[num].alive = 1;

    mysprites[num].size = size;

    mysprites[num].x = x;

    mysprites[num].y = y;

    mysprites[num].colormode = COLOR_256;

    mysprites[num].trans = MODE_TRANSPARENT;


    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;


    //initialize
    sprites[num].attribute0 = COLOR_256 | MODE_TRANSPARENT | y;


    switch (size)
    {
        case 8:  sprite_size = SIZE_8;  break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }


    sprites[num].attribute1 = sprite_size | x;
}
```

Another new function that is used in this program is SetTrans. This function allows you to selectively toggle the transparency flag of any sprite at runtime.

```
/////////////////////////////////////////////////////////
// Function: SetTransparency
// Changes the transparency of a sprite,
// MODE_NORMAL or MODE_TRANSPARENT
/////////////////////////////////////////////////////////
```

```
void SetTrans(int num, int trans)

{

    mysprites[num].trans = trans;

    sprites[num].attribute0 = mysprites[num].colormode |
        mysprites[num].trans | mysprites[num].y;

}


//////////////////////////////////////////////////////////

// Function: SetColorMode

// Changes the color mode of the sprite

// COLOR_16 or COLOR_256

//////////////////////////////////////////////////////////

void SetColorMode(int num, int colormode)

{

    mysprites[num].colormode = colormode;

    sprites[num].attribute0 = mysprites[num].colormode |
        mysprites[num].trans | mysprites[num].y;

}


//////////////////////////////////////////////////////////

// Function: WaitVBlank

// Checks the scanline counter for the vertical blank period

//////////////////////////////////////////////////////////

void WaitVBlank(void)

{

    while(*ScanlineCounter < 160);

}
```

The main function follows. Most of this code should look familiar to you after going through the previous sample programs, but there is some new code here that is needed to support alpha blending. In addition to the code for bouncing the sprites around on the screen is ■ section that toggles transparency on and off every so often.

```
//////////////////////////////////////////////////////////

// Function: main()

// Entry point for the program
```

```
//////////////////////////////////////////////////////////////////////

int main()
{
    int n;
    int counter = 0;
    int change = 0;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //load ball sprite
    for(n = 0; n < ball_WIDTH * ball_HEIGHT / 2; n++)
        SpriteData3[n] = ballData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image (512)
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH, 512);
        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }
```

```
//set transparency level
REG_BLDMOD = (1 << 4) | (1 << 10);
REG_COLEV = (8) + (8 << 8);


//main loop
while(1)
{
    //keep the screen civil
    WaitVBlank();

    //toggle transparency after an interval
    if (counter++ > 1000)
    {
        counter = 0;
        if (change)
        {
            change = 0;
            for (n = 0; n < NUMBALLS; n++)
                SetTrans(n, MODE_NORMAL);
        }
        else
        {
            change = 1;
            for (n = 0; n < NUMBALLS; n++)
                SetTrans(n, MODE_TRANSPARENT);
        }
    }


    for (n = 0; n < NUMBALLS; n++)
    {
        //update sprite x position
        mysprites[n].x += mysprites[n].dirx;
        if (mysprites[n].x > 239 - mysprites[n].size)
```

```
        {
            mysprites[n].x = 239 - mysprites[n].size;
            mysprites[n].dirx *= -1;
        }
        if (mysprites[n].x < 1)
        {
            mysprites[n].x = 1;
            mysprites[n].dirx *= -1;
        }


        //update sprite y position
        mysprites[n].y += mysprites[n].diry;
        if (mysprites[n].y > 159 - mysprites[n].size)
        {
            mysprites[n].y = 159 - mysprites[n].size;
            mysprites[n].diry *= -1;
        }
        if (mysprites[n].y < 1)
        {
            mysprites[n].y = 1;
            mysprites[n].diry *= -1;
        }


        //update the sprite properties
        MoveSprite(n);
    }
    //copy all sprites into object attribute memory
    UpdateSpriteMemory();
    }
}
```

Well, that's the end of TransSprite. Go ahead and run the program, and I'm sure you will agree it is fascinating to watch the sprites moving around with alpha blending enabled. There are so many things you can do with this—you are but limited by your imagination!

Rotation and Scaling

Another fascinating special effect that really makes sprites fun is the ability to rotate and scale them in real time. Is there really any need to draw prerotated sprites anymore when support for rotating a sprite is built into the GBA hardware? The process isn't perfect, because the GBA doesn't have a floating-point processor, so all rotation must be done with fixed-point math. But that can be solved easily enough with a precalculated array of sine and cosine values. This is a refinement over the SIN and COS arrays that you saw in the previous chapter, as there is no longer any need for a source file containing these radian values since they're just computed at the start of the program. This does cause a slight delay at the start of the program, but you could deal with that by displaying a splash screen and using the calculations as a sort of delay, so the player doesn't notice that an actual computational delay is taking place (and I'm talking about only a few short seconds). However, if your game sprites need to display shadows, or if you want more precision in the game's graphics, you will want to pre-rotate all sprite images. Some objects, however, where precision is not as important, such as with an asteroid or a missile, rotating in the game should work fine.

In order to use rotation and scaling, you must define a new struct that fills in the missing rotational elements of the OAM struct used previously. The new struct, RotData, points to the same address in OAM and might be thought of as a union struct. However, note the use of filler elements in the struct, followed by pa, pb, pc, and pd. These are new attributes that describe the sprite's behavior and are used for rotation and scaling.

```
typedef struct tagRotData

{

    u16 filler1;

    u16 pa;

    u16 filler2;

    u16 pb;      u16 filler3;

    u16 pc;      u16 filler4;

    u16 pd;

}RotData,*pRotData;
```

The declaration of a new pointer is needed to use this struct, and as you'll note, it points to the sprites struct array (defined earlier in the program).

```
pRotData rotData = (pRotData)sprites;
```

The only thing that needs explanation is the use of sine and cosine to actually rotate the sprites on the screen. As I mentioned in the previous chapter regarding background rotation, the GBA supports the rendering of a rotated sprite, but you must provide the

trigonometry for the actual rotation. That is accomplished with fixed-point (integer) math, which is a type of virtual floating-point emulation that is extremely fast. Calculations must be done with radians, so the usual 360 degrees must be converted to radians.

```
//math values needed for rotation

#define PI 3.14159265

#define RADIAN(n) (((float)n) / (float)180 * PI)
```

Here are the two SIN and COS arrays used to hold the precomputed angle of rotation values:

```
//precomputed sine and cosine arrays

signed int SIN[360];

signed int COS[360];
```

And here is the loop that creates the SIN and COS arrays of precomputed values. This is a somewhat time-consuming process that ties up the CPU for a few seconds, so I suggest displaying a splash or title screen before running this code.

```
for(n = 0; n < 360; n++)

{

    SIN[n] = (signed int)(sin(RADIAN(n)) * 256);

    COS[n] = (signed int)(cos(RADIAN(n)) * 256);

}
```

The RotateSprite Program

Now that you have some of the basics down for rotating sprites, it's time to write a sample program to demonstrate how it all works. I realize that I have skimmed over the material and that you may be wondering how it all works. Without listing the actual code beforehand and then explaining it, I think it makes more sense to just type in the actual program and see how it works *firsthand*. Figure 7.9 shows the output from the RotateSprite program.

The RotateSprite program clears out a lot of the code from the previous program in order to help you understand exactly what is going on just with the rotation and scaling of the sprite. Therefore, there is no background image. What I have done differently with this program is provide a means to control the sprite using the GBA's buttons. The LEFT, RIGHT, UP, and DOWN buttons will move the sprite on the screen; the A and B buttons rotate the sprite; while the L and R buttons change the scale of the sprite.

Create a new project in Visual HAM and call it RotateSprite. You may also load the project off the CD-ROM, located in \Sources\Chapter07\RotateSprite. The RotateSprite.gba file is the binary that you may run directly in VisualBoyAdvance (or another GBA emulator, if you wish).
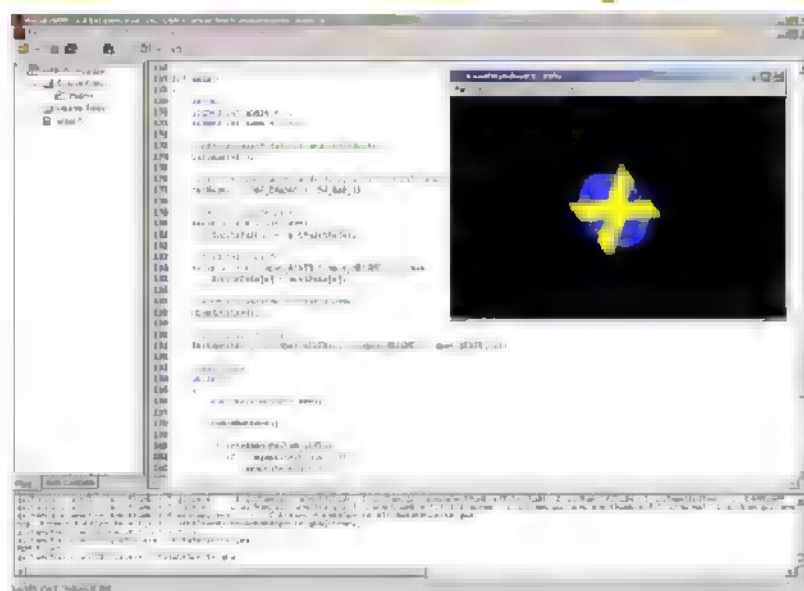
Figure 7.9

The RotateSprite program demonstrates how to rotate and scale a sprite from player input.

The RotateSprite Header File

The header file takes care of all the defines, includes, and so on and is to be typed into a new file called main.h. If you need help adding a new file to the project, refer to one of the previous projects in this chapter for a summary.

```
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// RotateSprite Project
// main.h header file
//////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include <math.h>
#include "spot.h"
#include "bg.raw.c"

//macro to change the video mode
```

```
#define SetMode(mode) REG_DISPCNT = (mode)


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)


//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
     (volatile unsigned short*)0x4000006;


//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)


//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)


//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;


//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)


//transparency stuff
#define REG_BLDMOD      *(unsigned short*)0x4000050
#define REG_COLEV       *(unsigned short*)0x4000052


//misc sprite constants
#define OBJ_MAP_2D              0x0
#define OBJ_MAP_1D              0x40
#define OBJ_ENABLE              0x1000
#define BG2_ENABLE              0x400
```

```
//attribute0 stuff
#define ROTATION_FLAG       0x100
#define SIZE_DOUBLE         0x200
#define MODE_NORMAL         0x0
#define MODE_TRANSPARENT    0x400
#define MODE_WINDOWED       0x800
#define MOSAIC              0x1000
#define COLOR_16            0x0000
#define COLOR_256           0x2000
#define SQUARE              0x0
#define TALL                0x4000
#define WIDE                0x8000


//attribute1 stuff
#define ROTDATA(n)          ((n) << 9)
#define HORIZONTAL_FLIP     0x1000
#define VERTICAL_FLIP       0x2000
#define SIZE_8              0x0
#define SIZE_16             0x4000
#define SIZE_32             0x8000
#define SIZE_64             0xC000


//Attribute2 stuff
#define PRIORITY(n)         ((n) << 10)
#define PALETTE(n)          ((n) << 12)


//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
```

```c
}Sprite,*pSprite;


typedef struct tagRotData
{
    u16 filler1;
    u16 pa;
    u16 filler2;
    u16 pb;
    u16 filler3;
    u16 pc;
    u16 filler4;
    u16 pd;
}RotData,*pRotData;


//create an array of 128 sprites equal to OAM
Sprite sprites[128];
pRotData rotData = (pRotData)sprites;


typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;
    int colormode;
    int trans;
    signed int rotate;
    signed int scale;


}SpriteHandler;


SpriteHandler mysprites[128];


//define the buttons
```

```
#define BUTTON_A 1

#define BUTTON_B 2

#define BUTTON_SELECT 4

#define BUTTON_START 8

#define BUTTON_RIGHT 16

#define BUTTON_LEFT 32

#define BUTTON_UP 64

#define BUTTON_DOWN 128

#define BUTTON_R 256

#define BUTTON_L 512


//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;


//keep track of the status of each button
int buttons[10];


//math values needed for rotation
#define PI 3.14159265
#define RADIAN(n) (((float)n)/(float)180 * PI)


//precomputed sine and cosine arrays
signed int SIN[360];
signed int COS[360];


#endif
```

The RotateSprite Source File

Here is the code listing for the main.c file of RotateSprite.

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// RotateSprite Project
```

```c
// main.c source code file
//////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"


//////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}


//////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
//////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
```

```
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}


//////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
//////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;


    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}
```

The InitSprite function has been changed again, this time providing support for rotation and scaling. I have highlighted key lines in bold text.

```
//////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
//////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;
    mysprites[num].rotate = ROTATION_FLAG;
    mysprites[num].scale = 1 << 8;
    mysprites[num].angle = 0;
```

```
        //in modes 3-5, tiles start at 512, modes 0-2 start at 0
        sprites[num].attribute2 = tileIndex;


        //initialize
        sprites[num].attribute0 = y |
            COLOR_256 |
            ROTATION_FLAG;


        switch (size)
        {
            case 8: sprite_size = SIZE_8; break;
            case 16: sprite_size = SIZE_16; break;
            case 32: sprite_size = SIZE_32; break;
            case 64: sprite_size = SIZE_64; break;
        }


        sprites[num].attribute1 = x |
            sprite_size |
            ROTDATA(tileIndex);
}


///////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
///////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(*ScanlineCounter < 160);
}
```

Here is the CalcAngles function, which generates the precomputed sine and cosine values for rotation:

```
///////////////////////////////////////////////////////////
// Function: CalcAngles
```

```
// Pre-calculates the sine and cosine tables
/////////////////////////////////////////////////////////
void CalcAngles(void)
{
    int n;
    for(n = 0; n < 360; n++)
    {
        SIN[n] = (signed int)(sin(RADIAN(n)) * 256);
        COS[n] = (signed int)(cos(RADIAN(n)) * 256);
    }
}
```

I didn't go over the RotateSprite function earlier, so now a short summary is called for. This function uses the precomputed SIN and COS arrays as if they were sin() and cos() functions, with the usual rotation algorithm. Since the SIN and COS arrays are filled with fixed-point integer values, with the decimal fixed between bits 8 and 9 (where the first ∎ bits represent the whole number, and the second 8 bits the fractional number), this code runs quite fast compared to floating-point code. After the new values have been calculated, they are plugged into the rotData structure for that particular sprite.

```
/////////////////////////////////////////////////////////
// Function: RotateSprite
// Rotates and scales a hardware sprite
/////////////////////////////////////////////////////////
void RotateSprite(int rotDataIndex, int angle,
    signed int xscale, signed int yscale)
{
    signed int pa,pb,pc,pd;

    //use the pre-calculated fixed-point arrays
    pa = ((xscale) * COS[angle])>>8;    pb = ((yscale) * SIN[angle])>>8;
    pc = ((xscale) * -SIN[angle])>>8;
    pd = ((yscale) * COS[angle])>>8;

    //update the rotation array entry
    rotData[rotDataIndex].pa = pa;    rotData[rotDataIndex].pb = pb;
```

```
        rotData[rotDataIndex].pc = pc;

        rotData[rotDataIndex].pd = pd;

}


//////////////////////////////////////////////////////////////

// Function: CheckButtons

// Polls the status of all the buttons

//////////////////////////////////////////////////////////////

void CheckButtons()

{

    //store the status of the buttons in an array

    buttons[0] = !((*BUTTONS) & BUTTON_A);

    buttons = !((*BUTTONS) & BUTTON_B);

    buttons = !((*BUTTONS) & BUTTON_LEFT);

    buttons = !((*BUTTONS) & BUTTON_RIGHT);

    buttons = !((*BUTTONS) & BUTTON_UP);

    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);

    buttons[6] = !((*BUTTONS) & BUTTON_START);

    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);

    buttons[8] = !((*BUTTONS) & BUTTON_L);

    buttons[9] = !((*BUTTONS) & BUTTON_R);

}


//////////////////////////////////////////////////////////////

// Function: Pressed

// Returns the status of a button

//////////////////////////////////////////////////////////////

int Pressed(int button)

{

    switch(button)

    {

        case BUTTON_A: return buttons[0];

        case BUTTON_B: return buttons;

        case BUTTON_LEFT: return buttons;
```

```
            case BUTTON_RIGHT: return buttons;

            case BUTTON_UP: return buttons;

            case BUTTON_DOWN: return buttons[5];

            case BUTTON_START: return buttons[6];

            case BUTTON_SELECT: return buttons[7];

            case BUTTON_L: return buttons[8];

            case BUTTON_R: return buttons[9];

        }

        return 0;

    }
```

The main function of the RotateSprite program is shown next. This function handles setting up the screen, calling the CalcAngles function to generate the SIN and COS lookup tables, and loads the sprite into OAM. After initialization, the program goes into a loop to handle button input and move the sprites on the screen.

```
///////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
///////////////////////////////////////////////////////////
int main()
{
    int n;

    //pre-calculate the SIN and COS tables
    CalcAngles();

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = spotPalette[n];

    //load ball sprite
    for(n = 0; n < spot_WIDTH * spot_HEIGHT / 2; n++)
```

```
            SpriteData[n] = spotData[n];


    //move all sprites off the screen
    HideSprites();


    //initialize the sprite at the center of the screen
    InitSprite(0, 120-spot_WIDTH/2, 80-spot_HEIGHT/2, spot_WIDTH, 0);


    //main loop
    while(1)
    {
        //comment out when running on real hardware
        for (n=0; n<1000; n++);
        //grab the button status
        CheckButtons();


        //control sprite using buttons
        if (Pressed(BUTTON_LEFT))
            if (--mysprites[0].x < 1)
                mysprites[0].x = 1;


        if (Pressed(BUTTON_RIGHT))
            if (++mysprites[0].x > 239-spot_WIDTH)
                mysprites[0].x = 239-spot_WIDTH;


        if (Pressed(BUTTON_UP))
            if (--mysprites[0].y < 1)
                mysprites[0].y = 1;


        if (Pressed(BUTTON_DOWN))
            if (++mysprites[0].y > 159-spot_HEIGHT)
                mysprites[0].y = 159-spot_HEIGHT;


        //buttons A and B change the angle
```

```
        if (Pressed(BUTTON_A))
            if (--mysprites[0].angle < 0)
                mysprites[0].angle = 359;


        if (Pressed(BUTTON_B))
            if (++mysprites[0].angle > 359)
                mysprites[0].angle = 0;


        //buttons L and R change the scale
        if (Pressed(BUTTON_L))
            mysprites[0].scale--;


        if (Pressed(BUTTON_R))
            mysprites[0].scale++;


        //update sprite position
        MoveSprite(0);


        //rotate and scale the sprite
        RotateSprite(0, mysprites[0].angle,
            mysprites[0].scale, mysprites[0].scale);


        //wait for vertical refresh before updating sprites
        WaitVBlank();


        //copy all sprites into object attribute memory
        //this is only possible during vertical refresh
        UpdateSpriteMemory();
    }
}
```

Animated Sprites

The only other special effect (or feature) of note aside from scaling, rotation, and alpha blending, would have to be animation. Many games on the GBA use rather static sprites, but

the graphically rich games always include animated sprites. From a programming perspective, animated sprites require a lot more memory than static sprites, because every frame of animation is another small bitmap image that must be kept in memory. The easiest way to animate a sprite is to copy a particular frame of an animation sequence into OAM so that it is rendered during the next screen refresh. From the perspective of OAM, there is just one sprite image, but your program copies a new version of the sprite bitmap into sprite display memory.

The AnimSprite Program

The AnimSprite program (shown in Figure 7.10) is very similar to the BounceSprite program, and even uses the same sphere image, only this version of the sphere includes many frames of animation so that it appears to be rotating in 3D.

The AnimSprite Header

Here is the header for the AnimSprite program, which should be typed into a file called main.h. If you haven't created the AnimSprite project yet, go ahead and create the project now, and add a new file, as you have done with other sample programs in this chapter.
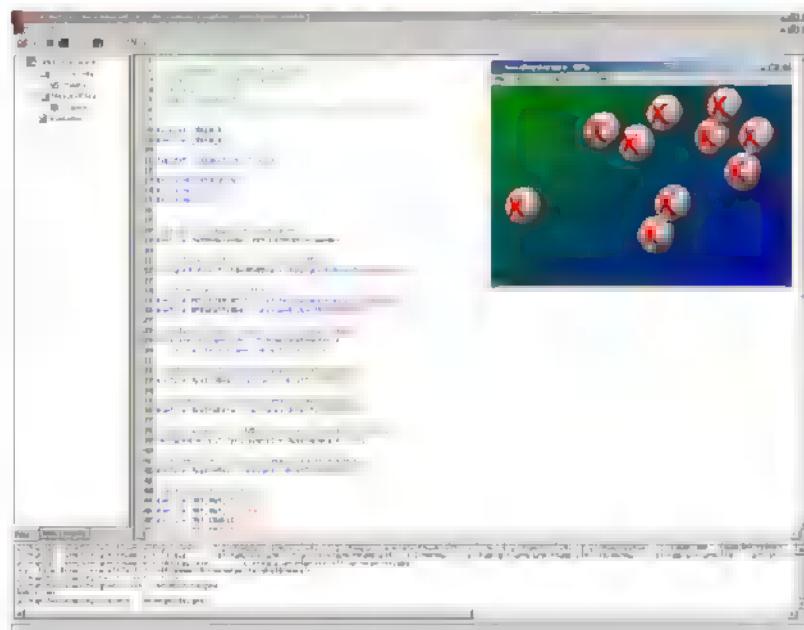
Figure 7.10

The AnimSprite program demonstrates how to draw animated sprites.

```
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
```

```
// AnimSprite Project
// main.h header file
/////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "bg.raw.c"
#include "ball2.h"


//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)

//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)
```

```
//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;


//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)


//misc sprite constants
#define OBJ_MAP_2D              0x0
#define OBJ_MAP_1D              0x40
#define OBJ_ENABLE              0x1000
#define BG2_ENABLE              0x400


//attribute0 stuff
#define ROTATION_FLAG           0x100
#define SIZE_DOUBLE             0x200
#define MODE_NORMAL             0x0
#define MODE_TRANSPARENT        0x400
#define MODE_WINDOWED           0x800
#define MOSAIC                  0x1000
#define COLOR_256               0x2000
#define SQUARE                  0x0
#define TALL                    0x4000
#define WIDE                    0x8000


//attribute1 stuff
#define SIZE_8                  0x0
#define SIZE_16                 0x4000
#define SIZE_32                 0x8000
#define SIZE_64                 0xC000


//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
```

```
        unsigned short attribute1;

        unsigned short attribute2;

        unsigned short attribute3;

}Sprite,*pSprite;


//create an array of 128 sprites equal to OAM

Sprite sprites[128];


typedef struct tagSpriteHandler

{

        int alive;

        int x, y;

        int dirx, diry;

        int size;


}SpriteHandler;


SpriteHandler mysprites[128];


#endif
```

The AnimSprite Source Code

Okay, now for the last source code listing of the chapter, the code for the main AnimSprite program. The code is similar to the BounceSprite program, so it should be familiar to you. Assuming you have already created the AnimSprite project, replace the default code in the main.c file with the following code listing.

```
/////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// AnimSprite Project
// main.c source code file
/////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
```

```c
#include "main.h"


#define NUMBALLS 10


/////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
/////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}


/////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
/////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}
```

```
/////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
/////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}


/////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
/////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;
```

```c
        switch (size)
        {
            case 8: sprite_size = SIZE_8; break;
            case 16: sprite_size = SIZE_16; break;
            case 32: sprite_size = SIZE_32; break;
            case 64: sprite_size = SIZE_64; break;
        }


        sprites[num].attribute1 = sprite_size | x;
}


////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}


void UpdateBall(index)
{
    u16 n;

    //load ball sprite
    for(n = 0; n < 512; n++)
        SpriteData3[n] = ballData[(512*index)+n];
}


////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////////////////////
```

```c
int main()
{
    int n;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
            COLOR_256, 512);

        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }

    int ball_index=0;

    //main loop
    while(1)
```

```
{
    if(++ball_index > 31) ball_index=0;
    UpdateBall(ball_index);


    for (n = 0; n < NUMBALLS; n++)
    {
        //update sprite x position
        mysprites[n].x += mysprites[n].dirx;
        if (mysprites[n].x > 239 - mysprites[n].size)
        {
            mysprites[n].x = 239 - mysprites[n].size;
            mysprites[n].dirx *= -1;
        }
        if (mysprites[n].x < 1)
        {
            mysprites[n].x = 1;
            mysprites[n].dirx *= -1;
        }


        //update sprite y position
        mysprites[n].y += mysprites[n].diry;
        if (mysprites[n].y > 159 - mysprites[n].size)
        {
            mysprites[n].y = 159 - mysprites[n].size;
            mysprites[n].diry *= -1;
        }
        if (mysprites[n].y < 1)
        {
            mysprites[n].y = 1;
            mysprites[n].diry *= -1;
        }


        //update the sprite properties
        MoveSprite(n);
```

```
        }

        //keep the screen civil
        WaitVBlank();

        //copy all sprites into object attribute memory
    UpdateSpriteMemory();

    }

}
```

Well, that sums up sprite rotation and animation! The only thing you need to worry about regarding the different video modes and backgrounds is that some backgrounds are not capable of being rotated or scaled, so if you come to a dead end and your own rotation code doesn't seem to be working, you might want to check the capabilities of the video mode and background you are using as a first attempt to get the program working. Another thing to remember is that the tile index for the sprite data is different for bitmap-based video modes (3-5) than for the tile-based modes (0-2). The reason for this is that bitmapped backgrounds require more video memory, and that encroaches on the sprite memory, so you must copy your sprite images and data into a higher position in sprite memory, depending on the video mode. Refer to the sample programs in this chapter for details on how to program the different modes when working with sprites.

Summary

Sprites are, without exception, the most important aspect of programming games on the GBA. This chapter has provided not only an overview and sample code for using hardware sprites, including how to convert source artwork into source code format, but this chapter has also delved into special effects. You have learned how to display and move sprites around on the screen using tile-based and bitmap-based video modes, as well as how to rotate and scale sprites in real time. This chapter also provided an explanation of sprite translucency using a process called alpha blending, as well as an example program that shows how to draw animated sprites on the screen.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: Two of the programs in this chapter featured no backgrounds, in order to make the programs easier to understand. Modify either the SimpleSprite or RotateSprite program, giving it a tile-based background.

Challenge 2: Test the BounceSprite2 program with various sprite sizes (8 x 8, 16 x 16, 32 x 32, and 64 x 64) to see how many sprites are available when using each of these sizes. Simply reduce NUMBALLS until all the balls are moving to find the value in each case.

Challenge 3: The TransSprite program looks pretty neat, don't you think? Well, you can do better, I'm sure! Modify the program so it uses two different types of sprites, and make each sprite animated. You can do this by simply loading two frames for each sprite and storing them consecutively in OAM. Simply determine how large each sprite is and index that far into OAM for each new sprite.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in Appendix D.

1. How many hardware sprites does the GBA support as an upper limit, regardless of video mode?
    - A. 64
    - B. 128
    - C. 256
    - D. 384

2. What is the maximum sprite size supported by the GBA?
    - A. 16 x 16
    - B. 32 x 32
    - C. 64 x 64
    - D. 128 x 128

3. What video modes support sprite rotation and scaling?
    - A. Mode 2
    - B. Mode 3
    - C. Mode 4
    - D. Any mode

4. What is the name of the define used to enable transparency in a sprite?
    - A. MODE_TRANSPARENT
    - B. MODE_ALPHABLEND
    - C. MODE_TRANSLUCENT
    - D. MODE_WINDOWED

5. What trigonometric functions are used to calculate degrees of rotation?
   A. cos and arctan
   B. sin and tan
   C. cosine and tangent
   D. sin and cos

6. True/False: Does the ARM7 processor have built-in support for floating-point numbers?
   A. True
   B. False

7. What special effects in all does the GBA provide for hardware sprites?
   A. Rotation and scaling
   B. Blitting, rotation, scaling, and alpha blending
   C. Blitting and scaling
   D. Blitting, rotation, and transparency

8. Where is the object attribute memory (OAM) image address located, where the actual sprites are stored?
   A. 0x6000000
   B. 0x7000000
   C. 0x6010000
   D. 0x4000052

9. What two programs were used in this chapter to convert sprite images into C source listings?
   A. gfx2gba and pcx2sprite
   B. bmp2gba and gif2gba
   C. pcx2gba and jpg2sprite
   D. pdf2sprite and doc2pdf

10. Which sprite attribute is used to control special effects, such as rotation, scaling, and transparency?
    A. Attribute0
    B. Attribute1
    C. Attribute2
    D. Attribute3

Part III

Meditating On

The Hardware

Welcome to Part III of *Programming The Nintendo Game Boy Advance: The Unofficial Guide*. Part III includes four chapters that are focused on low-level hardware programming, including interfacing with the Game Boy's buttons, using timers and synchronizing objects on the screen, programming the sound system, and interfacing with assembly language. A whole chapter is dedicated to ARM7 assembler, which is the lowest level possible, right down to the bare metal of the Game Boy Advance.

Chapter 8

Using Interrupts And Timers

Up to this point you have learned most of the concepts, theory, and code needed to write entire games for the GBA. Although each chapter in this book largely stands independently of the others, it would have been helpful to read them in order. The flip side to delving into the GBA's graphics system—without argument, the largest and most important aspect of GBA programming—so quickly is that you miss out on a lot of vital subjects such as interrupts, timers, and button input. These subjects are so vital that it is difficult to fully demonstrate the graphics system without them, and yet that would add ■ huge amount of complexity to the graphics code. Despite the discrepancy and apparent switch, these really are advanced subjects that would have been too difficult to explain in the first half of the book—so here we are!

This chapter will explain interrupts and how they work, how you can use them, and even how you can write them yourself. Then I talk about the all-important subject of timers, how to slow down your program to a consistent frame rate. This has been something of ■ problem in prior chapters (aside from using the vblank), but now you will have the means to correct it.

Have you have ever wondered how professional GBA programmers seem to have such fine control over the hardware? How their games just seem to run perfectly, smoothly, with accurate timing, consistent frame rates? I sure have! It has everything to do with this chapter, because these professional games are using interrupts and timers to keep the "machine" running smoothly. This subject will really help you to refine your GBA coding skills and make your code run smoothly and reliably.

Here are the subjects you'll learn about in this chapter:

- Using interrupts
- Using timers

Using Interrupts

Do you ever feel as if you are interrupted far too often when trying to get work done? I am constantly interrupted while writing code, writing the text of this chapter, and so on. There is ■ parallel in computerdom, and it takes the shape of either ■ hardware interrupt or a software interrupt. A hardware interrupt is a physical event that pauses the CPU while some other process (such as a memory copy) is occurring. For instance, a DMA memory copy causes ■ brief interrupt to occur, halting the CPU until it is finished. On the other hand, there are software interrupts, which are virtual interruptions of the program, all occurring within the CPU, rather than outside of it. A software interrupt is common in a multitasking operating system like Windows 2000 or XP. Since the GBA is a console video game machine, as you might have expected, all interrupts occur in the hardware side. The good news is that you can trigger one of these interrupts using a CPU or BIOS instruction.

An interrupt basically works like this. First, disable all interrupts, because if an interrupt occurs while you are screwing with the interrupt registers, your GBA could melt. Okay, not really, but it would probably look like your GBA is possessed because weird things could happen. In the Windows world, we call that a GPF, a general protection fault, meaning that the core has been corrupted. I have always thought of an operating system's core as the central armory in a medieval castle—the building inside the castle walls, surrounded by a courtyard, where merchants and farmers sell their goods.

Where was I? Oh yes, interrupts. After you have disabled interrupts, then you configure the interrupt registers before enabling the interrupts again. Think of it as telling the cannoneers atop your castle walls, "Don't you dare fire it while I'm reloading!"

The chief interrupt officer of the GBA is REG_IME, which has an unofficial title of "interrupt master enable register." When you want to disable interrupts, you set REG_IME = 0x0. Likewise, to enable interrupts, you set REG_IME = 0x1. If you simply set this register, nothing will happen, because you haven't specified which interrupts should occur—the who, what, when, where, and how, so to speak. To enable specific interrupts, you set specific bits in the interrupt enable register, REG_IE. This subordinate "enable interrupts" register works on each type of interrupt individually. There are interrupts available for DMA, vertical blank, horizontal blank, vertical count, timers, serial communications, and buttons, and each type of interrupt has a special bit value and a specific register. For instance, the interrupt register for DMA2 is REG_DMA2CNT, and the interrupt register for the display status is REG_DISPSTAT. Let's take a look at that one right now (see Table 8.1).

As I mentioned, REG_DISPSTAT is just the interrupt register for the display status, which is the most oft-used interrupt. Now, in order to actually turn on an interrupt, you need to know what REG_IE bits represent. Table 8.2 lists the bits for that register.

Table 8.1 REG_DISPSTAT Bits

Bit	Description
0	VB - vertical blank is occurring
1	HB - horizontal blank is occurring
2	VC - vertical count reached
3	VBE - enables vblank interrupt
4	HBE - enables hblank interrupt
5	VCE - enables vcount interrupt
6-15	VCOUNT - vertical count value (0-159)

Table 8.2 REG_IE Bits

Bit	Description
0	VB - vertical blank interrupt
1	HB - horizontal blank interrupt
2	VC - vertical scanline count interrupt
3	T0 - timer 0 interrupt
4	T1 - timer 1 interrupt
5	T2 - timer 2 interrupt
6	T3 - timer 3 interrupt
7	COM - serial communication interrupt
8	DMA0 - DMA0 finished interrupt
9	DMA1 - DMA1 finished interrupt
10	DMA2 - DMA2 finished interrupt
11	DMA3 - DMA3 finished interrupt
12	BUTTON - button interrupt
13	CART - game cartridge interrupt
14-15	unknown/unused

The REG_IE bits are used quite often and are needed in order to create any interrupt, so it is helpful to create some definitions of these bit values, as follows. The hexadecimal values in this list of definitions allow you to perform a bitwise AND with the REG_IE register in order to set the specific bit, without interfering with any of the other bits.

```
#define INT_VBLANK 0x0001

#define INT_HBLANK 0x0002

#define INT_VCOUNT 0x0004

#define INT_TIMER0 0x0008

#define INT_TIMER1 0x0010

#define INT_TIMER2 0x0020

#define INT_TIMER3 0x0040

#define INT_COM    0x0080

#define INT_DMA0   0x0100

#define INT_DMA1   0x0200

#define INT_DMA2   0x0400

#define INT_DMA3   0x0800

#define INT_BUTTON 0x1000

#define INT_CART   0x2000
```

It is pretty interesting how an interrupt actually occurs. What happens is that when an interrupt is triggered, the CPU saves the state of all the registers and then passes control to the interrupt service routine (which you must specify). After the ISR is finished, the CPU restores the registers and continues from the point where it was interrupted.

As for the ISR, that is something you must write yourself! Thankfully, it can be a C function, rather than assembler. The key to writing an ISR is understanding one simple fact: Every interrupt, regardless of type, is set to branch out to memory address 0x3007FFC. What you must do is intercept that memory address and have it point to your own ISR (a simple C function that I'll show you how to write). What I mean by "every interrupt" is exactly that, taken literally. All interrupts are passed to that memory address, so when an interrupt comes in, you must check to see which interrupt was triggered. The nice thing about this is that you need only write a single ISR for all the interrupts you are using in your GBA program.

There is another helper register called REG_IF that is a duplicate of REG_IE and is used to determine which interrupt was triggered (refer to Table 8.2 for the bit layout of REG_IF). However, don't be confused by this fact. One and *only* one interrupt will occur at a time! So the REG_IF register will have only one bit set, not several. You don't process all the interrupts that are occurring—that was a funny mistake I made when first learning about

interrupts. It makes perfect sense if you think about it. Since the CPU is saving everything and sending control off to 0x3007FFC, why would there be more than one interrupt happening?

The good news about that is that you can simply compare REG_IF with the various interrupt definitions to see which one is occurring. More than likely, you will be using just one or two interrupts in your own programs, so a comprehensive check for all the interrupts is not necessary. Just look for the interrupt you have turned on, and that is all. For example:

```
if ((REG_IF & INT_TIMER0) == INT_TIMER0)
{
    //your timer code goes here
}
```

At the end of your ISR, be sure to turn off the bit for that particular interrupt request:

```
REG_IF |= INT_TIMER0
```

Let's take this pseudocode a step further and make it a little more complete before actually writing a complete program. First, let's define a register to the memory address for interrupts:

```
#define REG_INTERRUPT *(unsigned int*)0x3007FFC
```

So now, all you have to do is pass the name of your ISR function to this register, and the compiler will copy the address of that function into the interrupt link. Here's a short snippet that includes all the steps:

```
//first, turn off interrupts
REG_IME = 0x00;
//make ISR point to my own function
REG_INTERRUPT = (unsigned int)MyHandler;
//turn on vblank interrupt
REG_IE |= INT_VBLANK;
//tell dispstat about vblank interrupt
REG_DISPSTAT |= 0x08;
//lastly, turn interrupts back on
REG_IME = BIT00;
```

Now all that is needed is your own function for dealing with interrupts. Since I called it MyHandler in the preceding code, that's what I'll call it here. I have not commented this function, so as to keep it short. The full-blown handler in the InterruptTest program (a little further on) fully explains each line.

```
void MyHandler(void)

{

    REG_IME = 0x00;

    Int_Flag = REG_IF;

    if((REG_IF & INT_HBLANK) == INT_HBLANK)

    {

        //horizontal refresh--do something quick!

    }

    REG_IF = Int_Flag;

    REG_IME = 0x01;

}
```

The InterruptTest Program

The InterruptTest program (shown in Figure 8.1) demonstrates how to create an interrupt service routine in the form of a callback function. It is surprisingly easy to set up custom interrupts on the GBA, so the program is fairly short. This program does something very simple, because I want you to focus more on the interrupt code than any fancy display code or demo. Therefore, this program simply draws a mode 3 pixel when an interrupt occurs. I should point out that horizontal blank is a very short time interval that you shouldn't screw around with, or the display could go fubar (for the scientist, that means the horizontal blank has been distended, resulting in possible loss of image). Drawing a pixel is ■ one-liner, but drawing a random pixel is a three-liner, so it provides just enough to prove the interrupt is working.
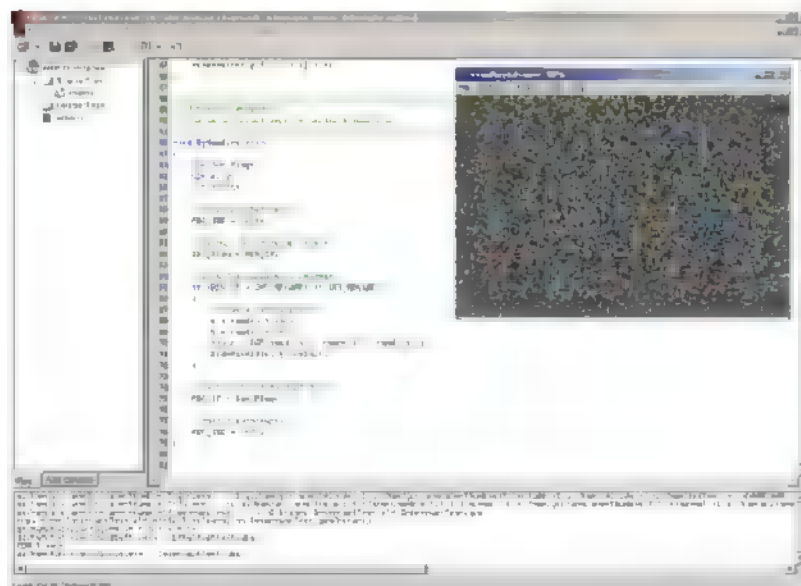
Figure 8.1

The InterruptTest program demonstrates how to create a custom interrupt service routine.

The benefit here also is that you gain some experience working with the hblank, which is different from vblank, because there are 160 hblank interrupts for every one vblank, so your hblank code must be fast! In this example, what is happening is that 160 pixels are being sent to video memory every time the screen is refreshed. Some pixels are added behind the scanline and don't appear until the next vblank, while some pixels are added before the scanline and do appear right away. It's an interesting thing to play around with. I would recommend against using hblank unless absolutely necessary because it affects the performance of the video system.

The InterruptTest Header File

```
//////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Interrupts, Timers, and DMA
// InterruptTest Project
// main.h header file
//////////////////////////////////////////////////////


#ifndef _MAIN_H
#define _MAIN_H


#include <stdlib.h>


//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;


//packs three values into a 15-bit color
#define RGB(r,g,b)  ((r)+(g<<5)+(b<<10))


//define some display registers
#define REG_DISPCNT *(u32*)0x4000000
```

```
#define BG2_ENABLE 0x400

#define SetMode(mode) REG_DISPCNT = (mode)


//define some interrupt registers
#define REG_IME          *(u16*)0x4000208

#define REG_IE           *(u16*)0x4000200

#define REG_IF           *(u16*)0x4000202

#define REG_INTERRUPT    *(u32*)0x3007FFC

#define REG_DISPSTAT     *(u16*)0x4000004


//create prototype for custom interrupt handler
void MyHandler(void);


//define some interrupt constants
#define INT_VBLANK 0x0001

#define INT_HBLANK 0x0002

#define INT_VCOUNT 0x0004

#define INT_TIMER0 0x0008

#define INT_TIMER1 0x0010

#define INT_TIMER2 0x0020

#define INT_TIMER3 0x0040

#define INT_COM 0x0080

#define INT_DMA0 0x0100

#define INT_DMA1 0x0200

#define INT_DMA2 0x0400

#define INT_DMA3 0x0800

#define INT_BUTTON 0x1000

#define INT_CART 0x2000


//create pointer to video memory
unsigned short* videoBuffer = (unsigned short*)0x6000000;


#endif
```

The InterruptTest Source File

Now for the main source code file of the InterruptTest program. This is a short program listing, thanks to the header file, allowing you to focus on exactly what is going on.

```c
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Interrupts, Timers, and DMA
// InterruptTest Project
// main.c source code file
//////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


#include "main.h"


//////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////
int main(void)
{
    //Set mode 3 and enable the bitmap background
    SetMode(3 | BG2_ENABLE);

    //disable interrupts
    REG_IME = 0x00;

    //point interrupt handler to custom function
    REG_INTERRUPT = (u32)MyHandler;

    //enable hblank interrupt (bit 4)
    REG_IE |= INT_HBLANK;

    //enable hblank status (bit 4)
```

```c
    REG_DISPSTAT |= 0x10;


    //enable interrupts
    REG_IME = 0x01;


    //endless loop
    while(1);
    return 0;
}


//////////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}


//////////////////////////////////////////////////////////////////
// Function: MyHandler
// Custom interrupt service callback function
//////////////////////////////////////////////////////////////////
void MyHandler(void)
{
    u16 Int_Flag;
    u16 x, y;
    u16 color;


    //disable interrupts
    REG_IME = 0x00;


    //backup the interrupt flags
    Int_Flag = REG_IF;
```

```
    //look for horizontal refresh

    if((REG_IF & INT_HBLANK) == INT_HBLANK)

    {

        //draw a random pixel

        x = rand() % 240;

        y = rand() % 160;

        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawPixel3(x, y, color);

    }


    //restore the interrupt flags

    REG_IF = Int_Flag;


    //enable interrupts

    REG_IME = 0x01;

}
```

Using Timers

The subject of timers is perhaps the most important subject you can learn about the GBA, aside from graphics programming, because timers are critical to keeping the game running at a stable frame rate, and they come in handy when you want to insert a delay into the game (for instance, when scrolling text on the screen). The timing within the GBA is precise. For example, the refresh rate (the time it takes to draw all 160 scanlines) takes 280,896 CPU cycles (also called ticks). The vertical blank period is not the same as vertical refresh—the blank is the period of time during which the pixel "pointer" (for lack of a better term) is moved from the bottom-right back up to the top-left to start refreshing the screen again—using the video buffer. This vblank period is exactly 83,776 cycles. To be more precise still, the horizontal blank (hblank) takes 228 cycles, while a horizontal draw (hdraw) takes 1,004 cycles. These are incomprehensible time periods for the human mind to grasp—billionths of a second, or nanoseconds.

There are four timers built into the GBA, and they are each capable of handling 16-bit numbers, meaning the timers count from 0 to 65,535. The timers are based on the system

clock. There are four frequencies available that you can set for the timers, as listed in Table 8.3.

Table 8.3 Timer Frequencies

Value	Frequency	Duration
0	16.78 MHz clock	Every 59.595 nanoseconds
1	64 cycles	Every 7.6281 microseconds
2	256 cycles	Every 15.256 microseconds
3	1,024 cycles	Every 61.025 microseconds

This table can be converted to a set of definitions to be used when setting up a timer:

```
#define TIMER_FREQUENCY_SYSTEM   0x0
#define TIMER_FREQUENCY_64       0x1
#define TIMER_FREQUENCY_256      0x2
#define TIMER_FREQUENCY_1024     0x3
```

There are a few things I need to go over about timers before you can create ▪ timer using one of these four frequencies. I'm sure you're eager to get started, so I'll be brief with the following descriptions. First, you will need to select one or more timers to program. The four timers have the following definitions, which point to the time control memory addresses:

```
#define REG_TM0CNT *(volatile u16*)0x4000102
#define REG_TM1CNT *(volatile u16*)0x4000106
#define REG_TM2CNT *(volatile u16*)0x400010A
#define REG_TM3CNT *(volatile u16*)0x400010E
```

Now these are merely the addresses of where to change the status bits for the timer. To actually read the values generated by the timers, you'll need to look at a different set of memory addresses set aside for this purpose. These are called the REG_TMxD addresses and are defined here:

```
#define REG_TM0D        *(volatile u16*)0x4000100
#define REG_TM1D        *(volatile u16*)0x4000104
#define REG_TM2D        *(volatile u16*)0x4000108
#define REG_TM3D        *(volatile u16*)0x400010C
```

The structure of these 16-bit memory addresses are laid out a bit at a time in Table 8.4.

Table 8.4 REG_TMxCNT Bits

Bit	Description
0-1	Frequency
2	Overflow from previous timer
3-5	*Not used*
6	Overflow generates interrupt
7	Timer enable
8-15	*Not used*

The first two bits are set to one of the TIMER_FREQUENCY_x defines above, while the other three options are set with the following defines:

```
#define TIMER_OVERFLOW          0x4
#define TIMER_IRQ_ENABLE        0x40
#define TIMER_ENABLE            0x80
```

Timers are quite a bit easier to use than interrupts because there is no callback function to worry about (although I would point out that it is entirely possible to create an interrupt of a timer). The TimerTest program is a very good demonstration of using timers, including the use of the overflow (which means that when one timer reaches the 65,536 limit, it resets to 0 and increments the next timer, if so configured). Overflow is a very nice feature in the GBA, providing for some convenient timing mechanisms, although you may use variables just as well to keep track of this sort of thing, perhaps by performing a test such as this:

```
timer = REG_TM0D;
if (timer % 65536)
{
    //overflow--time to deal with it
}
```

The TimerTest Program

And now to present the TimerTest program. Enjoy it while it lasts! Okay, the TimerTest program uses the font developed back in Chapter 5, "Bitmap-Based Video Modes," so you'll need the font.h file, which may be copied from the DrawText project folder from Chapter

5, or you may simply open the TimerTest project from \Sources\Chapter08\TimerTest. This program features both a header and source file, with the bulk of the GBA-specific hardware code hidden away in the header. Technically, this is a bad coding practice, but these programs are all so short, it would be silly to put just definitions, constants, and prototypes in a .h file, while moving all source listings into proper .c files. Therefore, the most commonly used functions are also included in the header.

That *is* the *proper* way to do it, after all, but it compiles all the same this way, so I prefer to keep things simple. Now, if you were to write you own GBA game, it would most likely be quite lengthy, so I would recommend using multiple source files for larger projects. You may even move graphics code into graphics.h and graphics.c, for instance, and then move the button code into button.h and button.c. It's entirely up to you—I present simple code listings, with each chapter and each project standing on its own so the reader may jump around at will, and leave organization up to you.

Now, about that TimerTest program. A screen shot is shown in Figure 8.2. Watch and learn.
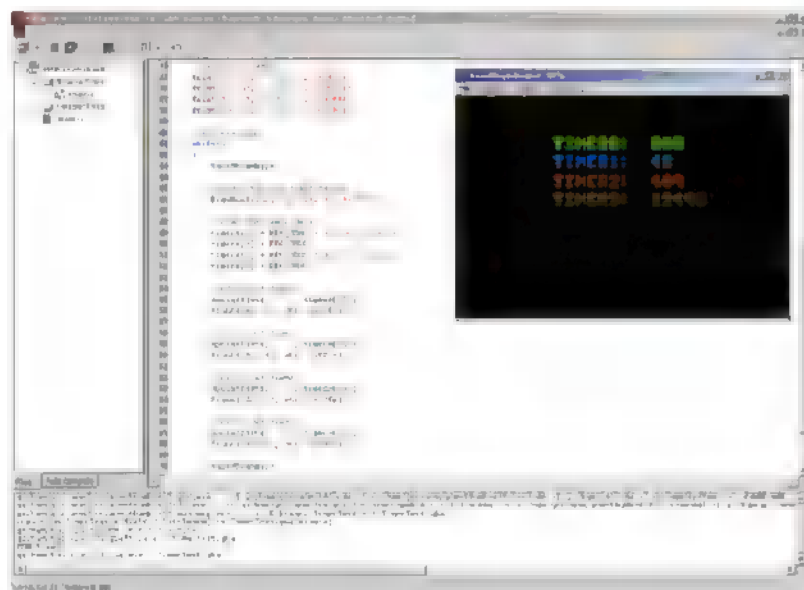
Figure 8.2

The TimerTest program displays four timers on the screen, two of which are overflows.

This project is like any other, with a main.c and main.h file. If you need a refresher on using Visual HAM, refer back to Chapter 3, "Game Boy Development Tools," for screen shots and a walk-through of creating projects and adding source files. The TimerTest header is listed first. Naturally, if you want to save some time, feel free to copy code from earlier projects and paste into each new project. I do reuse quite a bit of code from one project to the next.

The TimerTest Header

Here is the header for the TimerTest program, which should be saved in a file called main.h.

```
//////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// TimerTest Project
// main.h header file
//////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

#include <stdio.h>
#include <string.h>
#include "font.h"

//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawBox3(int, int, int, int, unsigned short)
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);

//define the timer constants
#define TIMER_FREQUENCY_SYSTEM 0x0
#define TIMER_FREQUENCY_64 0x1
#define TIMER_FREQUENCY_256 0x2
#define TIMER_FREQUENCY_1024 0x3
#define TIMER_OVERFLOW 0x4
```

```c
#define TIMER_ENABLE 0x80

#define TIMER_IRQ_ENABLE 0x40


//define the timer status addresses
#define REG_TM0CNT      *(volatile u16*)0x4000102

#define REG_TM1CNT      *(volatile u16*)0x4000106

#define REG_TM2CNT      *(volatile u16*)0x400010A

#define REG_TM3CNT      *(volatile u16*)0x400010E


//define the timer data addresses
#define REG_TM0D        *(volatile u16*)0x4000100

#define REG_TM1D        *(volatile u16*)0x4000104

#define REG_TM2D        *(volatile u16*)0x4000108

#define REG_TM3D        *(volatile u16*)0x400010C


//define some video mode values
#define REG_DISPCNT *(unsigned long*)0x4000000

#define MODE_3 0x3

#define BG2_ENABLE 0x400


//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


///////////////////////////////////////////////
// Function: Print
// Prints a string using the hard-coded font
///////////////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
```

```
        while (*str)
        {
            DrawChar(left + pos, top, *str++, color);
            pos += 8;
        }
    }


    ////////////////////////////////////////////////
    // Function: DrawChar
    // Draws a character one pixel at a time
    ////////////////////////////////////////////////
    void DrawChar(int left, int top, char letter, unsigned short color)
    {
        int x, y;
        int draw;

        for(y = 0; y < 8; y++)
            for (x = 0; x < 8; x++)
              {
                // grab a pixel from the font char
             draw = font[(letter-32) * 64 + y * 8 + x];
                // if pixel = 1, then draw it
                if (draw)
                    DrawPixel3(left + x, top + y, color);
              }
    }


    ////////////////////////////////////////////////
    // Function: DrawPixel3
    // Draws a pixel in mode 3
    ////////////////////////////////////////////////
    void DrawPixel3(int x, int y, unsigned short color)
    {
        videoBuffer[y * 240 + x] = color;
```

```
}

/////////////////////////////////////////////////////////////
// Function: DrawBox3
// Draws a filled box
/////////////////////////////////////////////////////////////
void DrawBox3(int left, int top, int right, int bottom,
    unsigned short color)
{
    int x, y;

    for(y = top; y < bottom; y++)
        for(x = left; x < right; x++)
            DrawPixel3(x, y, color);
}


/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}


#endif
```

The TimerTest Source Code

The main.c source code file for the TimerTest program is next. This code should be straightforward enough to follow, since the bulk of the program is stored away in the header file.

```
////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// TimerTest Project
// main.c source code file
////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


#include "main.h"


////////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////////
int main()
{
    char str[20];
    int timers;

    //switch to video mode 3
    REG_DISPCNT = (3 | BG2_ENABLE);

    //turn on timer0, set to 256 clocks
    REG_TM0CNT = TIMER_FREQUENCY_256 | TIMER_ENABLE;

    //turn on timer1, grab overflow from timer0
    REG_TM1CNT = TIMER_OVERFLOW | TIMER_ENABLE;

    //turn on timer2, set to system clock
    REG_TM2CNT = TIMER_FREQUENCY_SYSTEM | TIMER_ENABLE;

    //turn on timer3, grab overflow from timer2
```

```
REG_TM3CNT = TIMER_OVERFLOW | TIMER_ENABLE;

//print out labels
Print(70, 30, "TIMER0:", 0x0FF0);
Print(70, 40, "TIMER1:", 0xFE00);
Print(70, 60, "TIMER2:", 0x00FA);
Print(70, 70, "TIMER3:", 0x09B0);


//endless loop
while(1)
{
    WaitVBlank();

    //erase the old timer values
    DrawBox3(140, 30, 180, 80, 0x0000);

    //read the timer data
    timers[0] = REG_TM0D / (65536 / 1000);
    timers = REG_TM1D;
    timers = REG_TM2D / (65536 / 1000);
    timers = REG_TM3D;

    //print out timer0
    sprintf(str, "%i", timers[0]);
    Print(140, 30, str, 0x0FF0);

    //print out timer1
    sprintf(str, "%i", timers);
    Print(140, 40, str, 0xFE00);

    //print out timer2
    sprintf(str, "%i", timers);
    Print(140, 60, str, 0x00FA);
```

```
        //print out timer3

        sprintf(str, "%i", timers);

        Print(140, 70, str, 0x09B0);


        WaitVBlank();

    }


    return 0;

}
```

The Framerate Program

The Framerate program was a long time waiting. I have wanted to delve into timers since the fourth chapter in order to display what the GBA is capable of doing in the graphics department. I think this program demonstrates that the VisualBoyAdvance emulator is working perfectly, for one thing, because a consistent frame rate of 60 FPS comes through when vblank is used. On the flip side, the frame rate skyrockets out of control with vblank turned off! This means one thing—you definitely want to try your code once in a while without vblank to see how it's doing without any chains attached!

Figure 8.3 shows the Framerate program running with vblank turned off. This is basically the AnimSprite program from the previous chapter, which was the perfect example of a situation where knowing the frame rate would be extremely useful, as this was the most graphically intense program of the book so far. Look at that frame rate!
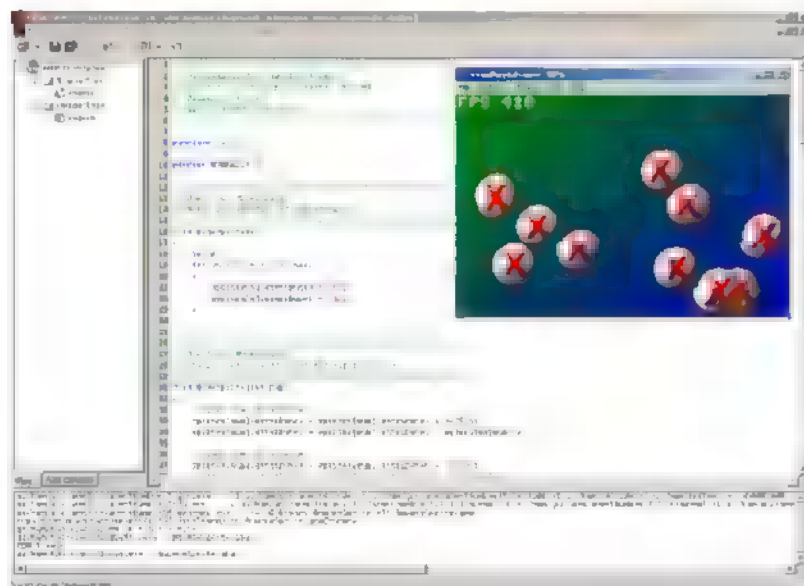
Figure 8.3

The Framerate program running with vblank turned off results in very high frame rates in both the emulator and an actual GBA. Note the image tearing, though!

Turning vblank back on (by uncommenting the WaitVBlank() line) results in a consistent 60 FPS (as shown in Figure 8.4), which is what you might expect under a video system that has ■ vertical refresh of 60 Hz. Now there are some weird things you can do to get around the 60 FPS limit without experiencing image tearing (as evidenced in Figure 8.3).
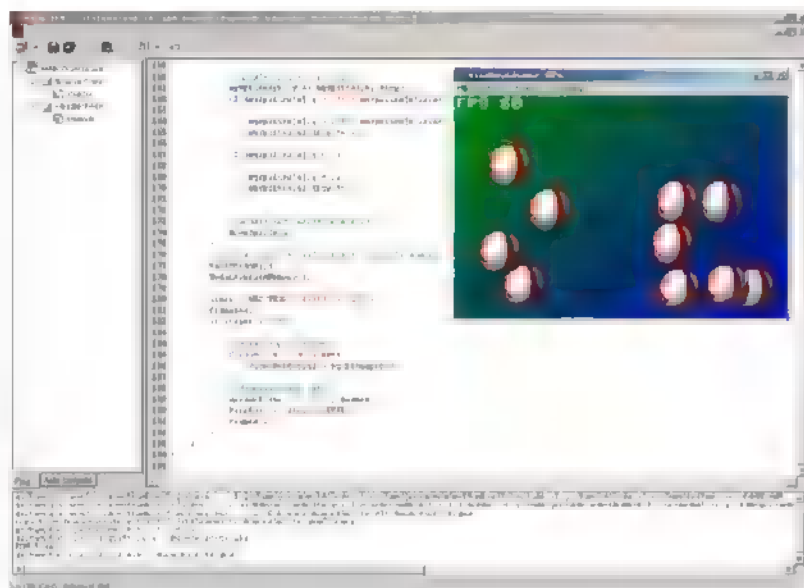
Figure 8.4

With vblank in use, the Framerate program reports a nice consistent 60 FPS, as expected, with nicely-drawn sprites.

This program is familiar if you have already worked through Chapter 7, "Rounding Up Sprites." If you are jumping around from chapter to chapter out of order, you'll have no problem running the program in this incarnation, because it is listed in its entirety. It might have been possible to just point out the differences between this Framerate program and the AnimSprite program from the last chapter, but there were significant changes to both the header and source code file, so I decided to just list both here in their entirety.

If you wish, you may load this project directly off the CD-ROM, which may be a good idea if you already worked through the AnimSprite project. It is located in \Sources\Chapter08\Framerate.

The Framerate Header

Now here is the header file, main.h, which is included by the main.c file. This file basically hides away all the messy details of a GBA program, allowing the main source code file, main.c, to stick to the goal and is also less distracting.

```
///////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// Framerate Project
// main.h header file
///////////////////////////////////////////////////////////
```

```c
#ifndef _MAIN_H
#define _MAIN_H

//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

#include <stdlib.h>
#include <stdio.h>
#include "bg.raw.c"
#include "ball.h"
#include "font.h"

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);
void WaitVBlank(void);

//define the timer constants
#define TIMER_FREQUENCY_SYSTEM 0x0
#define TIMER_FREQUENCY_64 0x1
#define TIMER_FREQUENCY_256 0x2
#define TIMER_FREQUENCY_1024 0x3
#define TIMER_OVERFLOW 0x4
#define TIMER_ENABLE 0x80
#define TIMER_IRQ_ENABLE 0x40

//define the timer status addresses
```

```
#define REG_TM0CNT       *(volatile u16*)0x4000102

#define REG_TM1CNT       *(volatile u16*)0x4000106

#define REG_TM2CNT       *(volatile u16*)0x400010A

#define REG_TM3CNT       *(volatile u16*)0x400010E


//define the timer data addresses

#define REG_TM0D         *(volatile u16*)0x4000100

#define REG_TM1D         *(volatile u16*)0x4000104

#define REG_TM2D         *(volatile u16*)0x4000108

#define REG_TM3D         *(volatile u16*)0x400010C


//macro to change the video mode

#define SetMode(mode) REG_DISPCNT = (mode)


//create a pointer to the video buffer

unsigned short* videoBuffer = (unsigned short*)0x6000000;


//define some video addresses

#define REG_DISPCNT *(volatile unsigned short*)0x4000000

#define BGPaletteMem ((unsigned short*)0x5000000)



//declare scanline counter for vertical blank

volatile u16* ScanlineCounter = (volatile u16*)0x4000006;


//define object attribute memory state address

#define SpriteMem ((unsigned short*)0x7000000)


//define object attribute memory image address

#define SpriteData ((unsigned short*)0x6010000)


//video modes 3-5, OAMData starts at 0x6010000 + 8192

unsigned short* SpriteData3 = SpriteData + 8192;
```

```
//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)


//misc sprite constants
#define OBJ_MAP_2D 0x0

#define OBJ_MAP_1D 0x40

#define OBJ_ENABLE 0x1000

#define BG2_ENABLE0x400


//attribute0 stuff
#define ROTATION_FLAG       0x100

#define SIZE_DOUBLE         0x200

#define MODE_NORMAL         0x0

#define MODE_TRANSPARENT    0x400

#define MODE_WINDOWED       0x800

#define MOSAIC              0x1000

#define COLOR_256           0x2000

#define SQUARE              0x0

#define TALL                0x4000

#define WIDE                0x8000


//attribute1 stuff
#define SIZE_8              0x0

#define SIZE_16             0x4000

#define SIZE_32             0x8000

#define SIZE_64             0xC000


//an entry for object attribute memory (OAM)
typedef struct tagSprite

{

    unsigned short attribute0;

    unsigned short attribute1;

    unsigned short attribute2;

    unsigned short attribute3;
```

```
}Sprite,*pSprite;


//create an array of 128 sprites equal to OAM
Sprite sprites[128];


typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;

}SpriteHandler;


SpriteHandler mysprites[128];


//////////////////////////////////////////////////
// Function: Print
// Prints a string using the hard-coded font
//////////////////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color);
        pos += 8;
    }
}


//////////////////////////////////////////////////
// Function: DrawChar
// Draws a character one pixel at a time
//////////////////////////////////////////////////
```

```c
void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            // grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            // if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
        }
}


//////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}


//////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
//////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}
```

```
#endif
```

The Framerate Source

Now for the main source code file for the Framerate program. You will likely see a lot of familiar code here, but there is also a lot of new code due to the use of timers to determine the frame rate. Most of the important code is located at the end of the main game loop, just after the WaitVBlank function call.

```c
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// Framerate Project
// main.c source code file
//////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


#include "main.h"


#define NUMBALLS 10


//////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
```

```
}


////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}


////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;
    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}


////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
////////////////////////////////////////////////////////////
```

```
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

    sprites[num].attribute1 = sprite_size | x;
}


/////////////////////////////////////////////////////////////
// Function: UpdateBall
// Copies current ball sprite frame into OAM
/////////////////////////////////////////////////////////////
void UpdateBall(index)
{
    u16 n;
    //copy sprite frame into OAM
```

```
        for(n = 0; n < 512; n++)
            SpriteData3[n] = ballData[(512*index)+n];
    }


    ////////////////////////////////////////////////////////////////
    // Function: main()
    // Entry point for the program
    ////////////////////////////////////////////////////////////////
    int main()
    {
        char str[10];
        int n;
        int frames = 0;
        int timer = 0;

        //set the video mode--mode 3, bg 2, with sprite support
        SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

        //draw the background
        for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

        //set the sprite palette
         for(n = 0; n < 256; n++)
            SpritePal[n] = ballPalette[n];

        //move all sprites off the screen
        HideSprites();

        //initialize the balls--note all sprites use the same image (512)
        for (n = 0; n < NUMBALLS; n++)
        {
            InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
                COLOR_256, 512);
```

```
            while (mysprites[n].dirx == 0)

                mysprites[n].dirx = rand() % 6 - 3;

            while (mysprites[n].diry == 0)

                mysprites[n].diry = rand() % 6 - 3;

    }


    int ball_index=0;


    //start the timer
    REG_TM3CNT = TIMER_FREQUENCY_256 | TIMER_ENABLE;



    //main loop
    while(1)

    {

        //increment the ball animation frame
        if(++ball_index > 31)ball_index=0;
        UpdateBall(ball_index);


        for (n = 0; n < NUMBALLS; n++)

        {

            //update sprite x position
            mysprites[n].x += mysprites[n].dirx;
            if (mysprites[n].x > 239 - mysprites[n].size)

            {

                mysprites[n].x = 239 - mysprites[n].size;
                mysprites[n].dirx *= -1;

            }

            if (mysprites[n].x < 1)

            {

                mysprites[n].x = 1;
                mysprites[n].dirx *= -1;

            }
```

```
            //update sprite y position
            mysprites[n].y += mysprites[n].diry;
            if (mysprites[n].y > 159 - mysprites[n].size)
            {
                mysprites[n].y = 159 - mysprites[n].size;
                mysprites[n].diry *= -1;
            }
            if (mysprites[n].y < 1)
            {
                mysprites[n].y = 1;
                mysprites[n].diry *= -1;
            }


            //update the sprite properties
            MoveSprite(n);
        }
        //copy all sprites into object attribute memory
        WaitVBlank();
        UpdateSpriteMemory();


        timer = REG_TM3D / (65536 / 1000);
        frames++;
        if (timer > 999)
        {
            //erase top of screen
            for(n=0; n < 2400; n++)
                videoBuffer[n] = bg_Bitmap[n];


            //display frame rate
            sprintf(str, "FPS %i", frames);
            Print(1, 1, str, 0xFFFF);
            frames = 0;
        }
```

Programming The Nintendo Game Boy Advance: The Unofficial Guide

```
    }

}
```

It's nice to know how the program is performing by displaying the frame rate, so I'm sure you'll find ■ use for this code in many a game. It could be optimized quite a bit, I won't deny it—particularly the code that erases the top of the screen. However, that doesn't seem to be affecting the frame rate at all. Really, from the screen shots, it is apparent that WaitVBlank is ■ serious detriment to the actual capabilities of the GBA! The CPU is capable of handling much more than these small example programs, which aren't even pushing the hardware. As soon as you start to see the frame rate drop below 60 FPS, then it's time to look into optimizations. But until then, don't waste time on it, and just keep working away on whatever game you are creating! After all, remember that optimization comes last, and only if it's needed. Worry more about writing good clean code first, and work on making an enjoyable game, because in all likelihood you won't tap the power of the GBA.

Summary

This chapter provided the theory and practical sides of using interrupts and timers to enhance your GBA programs. Using these two key hardware facilities, you will be able to better control the granularity of your programs—that is, how fast or slow they run, how smoothly the screen is refreshed, and how to process code based on certain interrupts. Not only did you learn how to use interrupts and timers, you have learned the practical use for them by writing an animated sprite program that displays the frame rate. As this is an extremely useful new feature, I'm sure you'll find a need for it in all of your own GBA projects.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: The InterruptTest program draws a pixel every time during every hblank. Modify the program so it instead draws a box during every vertical blank instead.

Challenge 2: The TimerTest program displays the values of each of the four timers, two of which are set to specific frequencies, the other two set to overflow. Modify the program using different sets of frequencies and note the change in the timers displayed on the screen.

Challenge 3: The Framerate program displays a consistent 60 FPS. Bump up the number of sprites displayed by modifying the NUMBALLS constant, adding 10 balls to the number each time, and note the change in the frame rate as the ball count increases. Try to determine

the maximum number of balls that can be animated before the frame rate drops below 60 FPS.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. What is the interrupt master enable register that turns interrupts on or off?
   - A. REG_IE
   - B. REG_IF
   - C. REG_IME
   - D. REG_DISPSTAT

2. How many interrupts are available on the GBA?
   - A. 14
   - B. 4
   - C. 8
   - D. 12

3. What register is used to enable interrupt status for vblank, hblank, and vcount?
   - A. REG_IME
   - B. REG_DISPSTAT
   - C. REG_INTERRUPT
   - D. REG_CODE

4. To what memory address does the CPU shift control during an interrupt?
   - A. 0x6000000
   - B. 0x7001000
   - C. 0x4F00401
   - D. 0x3007FFC

5. What register is used in a custom interrupt callback function to determine which interrupt has occurred?
   - A. REG_IF
   - B. REG_IE
   - C. REG_DISPCNT
   - D. REG_IME

6. Which interrupt, if enabled, is triggered 160 times for every screen refresh?
   - A. INT_DMA1
   - B. INT_HBLANK

C. INT_TIMER3
        D. INT_VBLANK

7. True/False: Does the LCD screen on the GBA handle screen refresh itself?
        A. True
        B. False

8. How many CPU cycles are used up for every vertical refresh of the screen?
        A. 280,896
        B. 4,836,938,238
        C. 160
        D. 16,386

9. How many timers are available in the GBA?
        A. 1
        B. 2
        C. 3
        D. 4

10. What is the largest numeric value that a 16-bit timer can handle?
        A. 4,096
        B. 16,384
        C. 65,536
        D. 1,048,576

Chapter 9

The Sound System

This chapter covers the fascinating subject of sound playback on the Game Boy Advance, with coverage of the sound hardware, digital sound formats, and the code to play sound samples. There are two sample programs in this chapter that show how to use the sound hardware on the GBA, from a simple playback program to a more elaborate program that uses button input to play several sounds. Are you ready to jump into the code and get started? This is a pretty fast-paced chapter that gets down to the metal and shows you exactly what you need to play sound. You will be adding sound to your own games in no time.

Here are the main topics of this chapter:

- Introduction to sound programming
- Playing digital sound files
- The PlaySamples program

Introduction to Sound Programming

The GBA has ■ very good sound system that was well designed and is perfectly suitable for handheld games. While most gamers just use the built-in speaker, the GBA does support stereo sound when using headphones. This is due to the dual digital channels in the sound chip. Ideally, you will want to wear headphones while playing games in order to take advantage of stereo sound, because the built-in speaker simply combines the two channels, dropping the stereo effect. If you have not tried it with headphones, you'll be surprised by how much better the games are in stereo.

GBA Sound Hardware

The GBA has two 8-bit digital-to-analog converters (DACs) for playing digital sound effects and music. These two channels, which are referred to as direct sound, support 8-bit signed samples. In addition, the GBA is backward compatible with previous Game Boy models, so it includes the earlier four sound channels. The two channels are called Direct Sound A and Direct Sound B and are capable of playing back 8-bit signed PCM samples. Pulse code modulation (PCM) is a raw format that is supported by most sound editor programs and may be saved as a .wav file.

FM Synthesis Support

The sound chip is backward compatible with Game Boy Color, providing four FM sound channels. Frequency modulation (FM) synthesis is a method of alternating the frequency of ■ sine wave at fast intervals to produce sound effects and music. The result is not bad, but how can I explain the output? It sounds fuzzy, like there is white noise in the sound, as in an improperly tuned radio station or TV channel—quite different from digital sound. Since I can't imagine any practical use for the four FM channels in the GBA, throwbacks to a previous decade, I am going to focus exclusively on the two direct sound channels. I hope you understand my reasoning, because there is no need for FM synthesis when you have ■ DAC available! That is akin to preferring an Apple II over a Pentium 4 PC—without considering the novelty, that is. For all practical purposes, just ignore the compatibility sound channels on the GBA, and focus on digital sound. There is no comparison.

However, I don't want to dismiss FM sound completely, because there are cases where it can be helpful, in some circumstances where digital sound is overkill. For instance, FM is great for doing some kinds of sound effects, such as an airplane engine (which runs continuously), or for the sound of wind perhaps.

Using Direct Sound for Digital Playback

Frequency modulation does work well to simulate sound mixing and does sound pretty good, in the context of small handheld games. In comparison, though, FM is simulated sound, rather than real sound. The reason for the poor sound quality in earlier Game Boy models was that they did not have the luxury of a DAC (whereas the GBA has two of them). With the GBA you can create sound effects yourself using a wave editor tool like Syntrillium's Cool Edit 2000 (included on the CD-ROM under \Tools\Cool Edit 2000), or you can download some

public domain wave files off the Web to use in your games. There is a tool for converting a wave file to ■ C source file and then storing the wave file's bytes inside ■ C array, in the same manner that bitmaps are converted. I will show you how to do it a little later in this chapter. Take ■ look at Figure 9.1 for an illustration of a waveform.
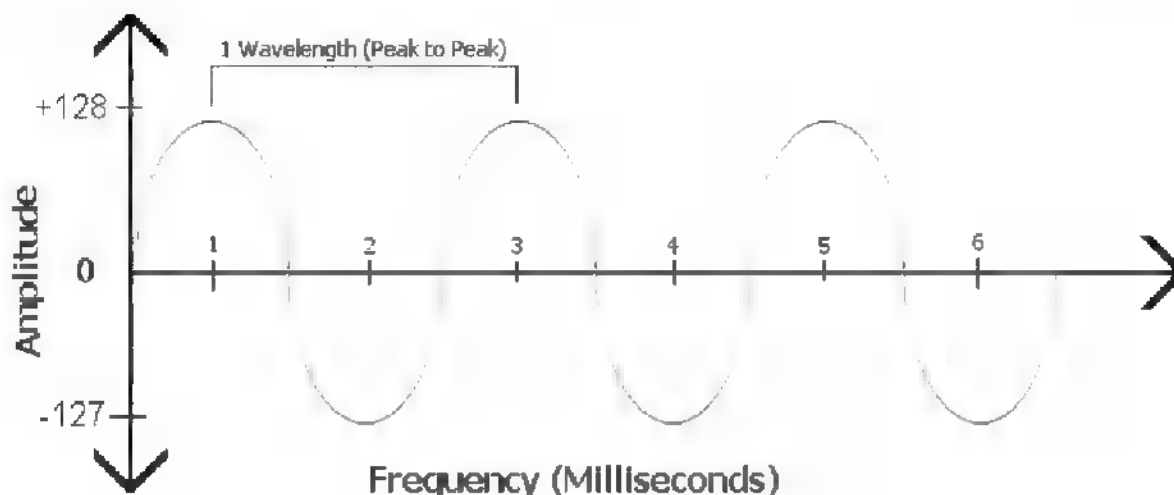
Figure 9.1 - The sound produced by a waveform is

determined by frequency and amplitude.

Sound Mixing

There is one issue with the GBA's sound capabilities, and it's not a problem at all once you understand the sound hardware. The GBA is capable of only a single digital sound at a time (per channel), with no support for mixing. What at first seems to be a problem, however, is really only the norm. Your PC doesn't have a hardware sound mixer either! Of course, ∎ PC sound card can output CD music along with digital sound produced by a program (such as Windows Media Player or WinAmp), that level of hardware mixing does not translate to games at all. Indeed, the latest fast-paced game for the PC must do sound mixing on its own, as that is not built into the PC. Now, before you object, what I mean is that the *game engine* (such as direct sound) does the sound mixing, which for all practical purposes is a function of the game, while DirectX just happens to be installed separately.

The GBA has a very good sound chip built in that is at least on par with the early PC sound cards, which is fantastic for a handheld—while lacking such obvious things as Dolby™ Pro-Logic™, Dolby DTS™, Dolby Surround Sound™. I hope that earned a chuckle, because obviously such support is useless coming through the built-in speaker or headphones. The dual digital channels on the GBA are perfect for the types of games developed for it. Fortunately, HAM comes with an excellent sound-mixing library called Krawall.

Krawall is a complete sound engine for the GBA, providing everything you will need for a complete game sound solution, with an emphasis on speed, high-quality playback, and a

straightforward API. Although HAM includes the free version of Krawall, I encourage you to peruse the Krawall Web site at http://mind.riot.org/krawall and download the latest version with documentation and learn how to use it.

Krawall is free for personal use but does require a license for commercial use. Also, the free version is not as powerful as the fully licensed version. If you are serious about GBA sound, then you need to get at least a personal licensed copy for your own use and should purchase ■ commercial license without a second thought if you are an officially licensed GBA developer. In addition to mixing wave samples, Krawall also features a ProTracker module player that can play .mod, .xm, and .s3m music files flawlessly in the background while playing sound effects in the "foreground."

Building a sound mixer is somewhat beyond the goals of this single chapter, so I encourage you to look into Krawall as a solution. There are other sound libraries available for the GBA, which I have listed in Appendix B, "Recommended Books and Web Sites."

Playing Digital Sound Files

Digital playback on the GBA is somewhat involved to the uninitiated, but the actual source code is not difficult to write and is definitely manageable in 20-30 lines of code. I will go over the specifics of the sound system and describe the registers and defines you will need to write ■ sound playback function. For starters, I'll walk you through ■ simple SoundTest program, which plays a single sample and ends. After you have gained an understanding of simple playback, I'll show you a program called PlaySamples, which plays several sounds based on button input.

Playing Digital Sounds

The key to sound playback on the GBA is not getting the sound going, but how to make it stop when the sample is finished. The GBA doesn't inherently know when it has reached the end of a sample. There are two methods of controlling playback: DMA and interrupts. DMA is the preferred method, because it doesn't require any intervention from the programmer. Once the sample is started, the DMA controller automatically feeds the sound buffer. Interrupt-driven sound, on the other hand, requires the programmer to feed the sound buffer at each interval (which is usually during the vblank).

Wave samples for a GBA game must be converted to a C array and should be loaded consecutively into the direct sound memory buffer called <u>REG_FIFO_A</u>, which starts at address 0x40000A0. Granted, there are always other ways to solve a problem, and in the case of GBA sound, you could convert a wave file to a binary file and link it into the program, although it's not a great solution when you are just learning this material.

Sound playback is interesting on the GBA. The 8-bit signed sound samples (which have ■ value range of only - 127 to 128) are played back by the GBA sound chip in a first-in first-out (FIFO) process, meaning that lower address bytes are played first. While it might sound at first that sound playback goes in reverse, that is only a matter of how you perceive memory. I perceive memory being laid out linearly, 1 byte at ■ time, down ■ very long line (per-

haps like a train).

Some imagine computer memory in a 2D or even 3D layout, but I submit that the linear analogy more closely resembles the actual state. It is true that a memory chip is filled with nanoscopic transistors, or gates, in a grid and is even layered, so the perception of a 2D or 3D memory chip is accurate from a hardware perspective. However, we aren't electrical engineers—or at least, I'm not!—so it is the software standpoint that matters here. The one-dimensional line analogy will help you to better understand how software works, if you have never really thought about it in detail. A wave file or bitmap file converted to a C array is a linear array of bytes, which might be thought of as pure bits. Now, a sound sample comprises just 1 signed byte, but it is the linear playback of many sample bytes, sent through the DAC, that produces a digital sound.

There are two possible ways to play a sample: using either DMA or an interrupt. DMA mode is more efficient because consecutive samples are automatically loaded without interruption to the game. The interrupt mode must briefly pause the program to load the FIFO buffer but is possibly easier to use, and perhaps even necessary to use, in some cases.

The direct sound channels are controlled by the REG_SOUND_CNT_H register located at memory address 0x04000082, which has the layout shown in Table 9.1.

Table 9.1 REG_SOUND_CNT_H Bits

Bits	Description
0-1	Channel 1-4 volume control
2	Direct Sound A volume control
3	Direct Sound B volume control
4-7	*Unused*
8	Enable Direct Sound A to right speaker
9	Enable Direct Sound A to left speaker
10	Direct Sound A sampling rate timer select
11	Direct Sound A reset FIFO
12	Enable Direct Sound B to right speaker
13	Enable Direct Sound B to left speaker
14	Direct Sound B sampling rate timer select
15	Direct Sound B reset FIFO

Since you will need a list of defines in order to program the sound channels, I'll provide that a little prematurely right now:

```
#define SND_ENABLED              0x00000080
#define SND_OUTPUT_RATIO_25      0x0000
#define SND_OUTPUT_RATIO_50      0x0001
#define SND_OUTPUT_RATIO_100     0x0002
#define DSA_OUTPUT_RATIO_50      0x0000
#define DSA_OUTPUT_RATIO_100     0x0004
#define DSA_OUTPUT_TO_RIGHT      0x0100
#define DSA_OUTPUT_TO_LEFT       0x0200
#define DSA_OUTPUT_TO_BOTH       0x0300
#define DSA_TIMER0               0x0000
#define DSA_TIMER1               0x0400
#define DSA_FIFO_RESET           0x0800
#define DSB_OUTPUT_RATIO_50      0x0000
#define DSB_OUTPUT_RATIO_100     0x0008
#define DSB_OUTPUT_TO_RIGHT      0x1000
#define DSB_OUTPUT_TO_LEFT       0x2000
#define DSB_OUTPUT_TO_BOTH       0x3000
#define DSB_TIMER0               0x0000
#define DSB_TIMER1               0x4000
#define DSB_FIFO_RESET           0x8000
```

The sound hardware is quite helpful when it comes to actually playing the sound sample. All you have to do (after copying the sample into the appropriate memory address for playback) is tell Direct Sound A or B to watch a specific timer for an overflow. You learned about timers in the previous chapter, which was kind of convenient, right? Well, it was planned that way. <Smile.> If you skipped over Chapter 8, "Using Interrupts and Timers," I recommend that you go back to it and first learn how timers and interrupts work before proceeding any further into this chapter.

When the specified timer (0 or 1) overflows, Direct Sound A or B will send another byte from the FIFO to the DAC for playback. The key is setting up the timers to the specific sampling rate of the wave file so it sounds right. Remember that the timers are used to play back the sound at the correct rate to accurately reproduce the sound. The way you can determine how to set the timers is by calculating how often a sample should be sent to the DAC, and this is based on the CPU cycles. At 16.7 MHz, the CPU has 16,777,216 cycles per second; this is a fixed value that you can count on in your GBA games, because the architecture of ■ console is fixed. To determine the number of cycles per sample, simply divide 16,777,216 by the sampling rate.

For example, suppose you want to play back a sample at 44.1 kHz, which is CD-quality music. Granted, this would take an enormous amount of memory, and you wouldn't want to do this in practice, so let's just call this a hypothetical situation. 16,777,216 / 44,100 = 380.44, so you would want to set a timer to send a sample to the DAC every 380 CPU cycles. A sampling rate of one-fourth CD quality is more realistic for GBA sounds, so let's calculate

it. 16,777,216 / 11,025 = 1,521.74, or rather, 1,521 cycles per sample. As you can see, the sound playback itself doesn't require much of the CPU's time, although the overhead of using timers and copying samples into memory does take a few more cycles.

Now, how would you go about setting up a timer to send a sample to the DAC every 1,521 CPU cycles? The timers are 16-bit, meaning they have a range of up to 65,535, with a selectable frequency of 1, 64, 256, or 1,024 CPU cycles. To program the counter, which will return to the preset value after an overflow, simply subtract the cycles from 65,535 to set the initial value of the timer. So, an 11 kHz sample would require a timer set to 65,535 - 1,521 = 64,014.

The SoundTest Program

Sound programming is not easy to explain as it is, and when dealing with registers and binary numbers, it can be quite confusing. To put this all into perspective, I've written a program called SoundTest, which simply plays a sound sample called *splash*, which is converted from splash.wav to splash.c and included in the main.c file. The program is only about a page in length, and the actual sound code is 20 or so lines. Figure 9.2 shows the program, although there is nothing displayed on the screen, as this program simply outputs the sound and then ends.
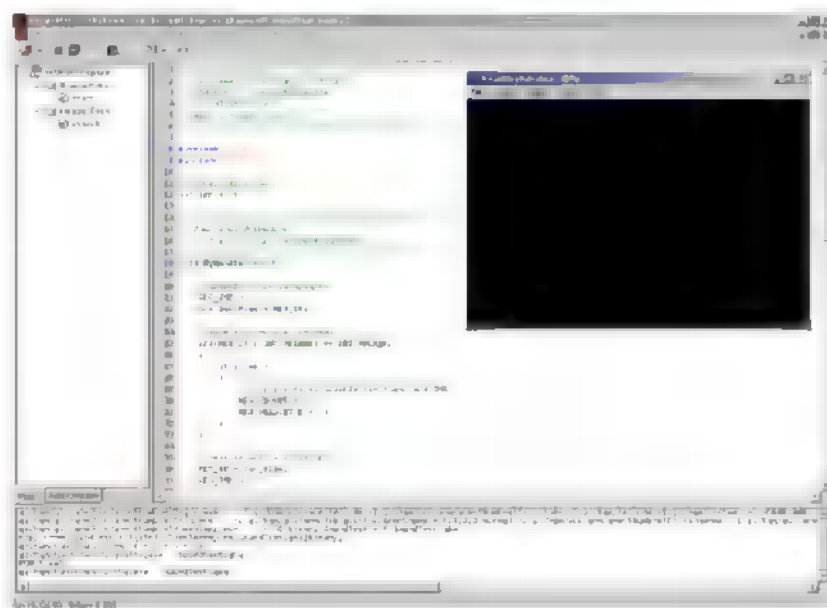
Figure 9.2
The SoundTest program plays a digital sound sample.

Converting the Sound File

In order to play a digital sample, a wave file must be converted into the raw binary format that the GBA can recognize and use. As explained earlier, that format is PCM and is stored in a .wav file. You are free to use any sound editing or converting program you like, but I prefer Cool Edit 2000. Some of the .wav files that I have are not all in PCM format. Here are some of the wave formats (or rather, audio codecs) that you are likely to encounter:

- A/mu-Law Wave

- ACM Waveform

- DVI/IMA ADPCM

- Microsoft ADPCM

- Windows PCM

As long as your sound-editing software is able to save files in the Windows PCM format, then the GBA will be able to play it. If it's not in the correct format, the converter program will print out an error message and fail to convert the file.

The program I have used to convert a wave file for use on the GBA is wav2gba, written by Rafael Vuijk (a.k.a. Dark Fader), and may be downloaded from http://darkfader.net/gba. I have included the wav2gba.exe program in each of the project folders for this chapter under \Sources\Chapter09 on the CD-ROM, as well as in the \Tools folder. The wav2gba program is a command-line tool, just like the gfx2gba program you have used to convert graphics in previous chapters. Wav2gba has this syntax:

```
wav2gba <input.wav> <output.bin>
```

If you open a Command Prompt window (Start, Programs, Accessories menu), change to the folder for the SoundTest program (using the CD command), which is located in \Sources\Chapter09\SoundTest on the CD-ROM. You will of course want to copy the sources off the CD-ROM to your hard drive and then remove the read-only property from \Sources and all subfolders and files (simply right-click on \Sources and select Properties, then uncheck the Read Only check box).

Assuming you are in the SoundTest folder, here is the command you would type in to convert the splash.wav file:

```
wav2gba splash.wav splash.bin
```

This will create a file called splash.bin in the current folder. Unfortunately for us, the splash.bin file is not exactly in the most useful format. What we need instead is a splash.c file with ∎ byte array containing the splash.wav sample. The splash.bin file could be linked into the program via an assembler file or converted into an .elf file and linked into the .exe, but that is a lot more difficult than simply using ∎ source code file (although I know some would not agree with me on that point). So, what is needed is a program to convert ∎ raw binary file into a generic C source file containing an array of bytes. There is a program that is perfect for the job, called bin2c.exe, also written by Dark Fader. The syntax of this program is also very simple:

```
bin2c <input.bin> <output.c>
```

To convert the splash.bin file, you can simply use the bin2c program like this:

```
bin2c splash.bin splash.c
```

The resulting file looks like this (truncated for space):

```
const unsigned char splash[] =
{
0x00,0x00,0x00,0x00,0x00,0x00,0x7D,0x00,0x00,0x00,0x00,0x00,
0x18,0x2D,0x00,0x00,0xFA,0x00,0xFD,0xFD,0x03,0x00,0xFD,0x03,
0xFD,0x03,0x00,0xFA,0xFD,0xFD,0xFA,0xFD,0x03,0x00,0x00,0x03,
0x03,0x0C,0x00,0x06,0x00,0x03,0x03,0x00,0x00,0xFD,0xFA,0xFD,
. . .
0xFA,0xFD,0x00,0xFA,0x06,0x00,0xFD,0xFD,0xFD,0xF7,0x00,0xFD,
0xFD,0x03,0xFD,0xFA,0x03,0xFA,0x03,0x00,0xFD,0x03,0x09,0xFD,
0x00,0x00,0xFA,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

Since I use these two programs to convert waves so often, I wrote a short batch file, called wav.bat, that will convert a wave file to a C file, like this. You can use Notepad to write this short batch program:

```
@echo off
wav2gba %1.wav %1.bin
if exist %1.bin bin2c %1.bin %1.c
```

Now, instead of calling on two programs with a total of four parameters (which is ■ lot of typing when you need to convert a bunch of waves!), I simply type this:

```
wav splash
```

The batch file calls on wav2gba and bin2c to convert the wave to a C file. Simple!

I should mention something about conversion errors you are likely to encounter, some of which are obscure. There is one error that reads like this:

```
'data' not found
```

Another error message looks like this:

```
8 bit required
```

Both error messages are related to an unsupported wave file format. The files must be saved in a PCM wave format, so just load up a wave you are having trouble with into Cool Edit 2000 or a similar sound-editing tool, and then do a Save As to convert it to a PCM wave. That should take care of the problem. In some cases, you may also need to downsample the wave from 16 bits to 8 bits, because the wav2gba program is smart enough to know that only 8-bit samples will work on the GBA and will refuse to convert 16-bit samples. You will need to downsample those files. In Cool Edit 2000, you can do this from the Edit menu by selecting Convert Sample Type, or by simply pressing F11 to convert the wave.

Once you have converted a wave file, you need to keep the intermediate splash.bin file handy because it is a raw binary file, and you'll need to know the exact file size in bytes in order to plug that value into the SoundTest program. If you are still in the Command Prompt window from converting the file, you can get a list of files by typing "DIR" and taking note of the size of splash.bin. Otherwise, you'll have to right-click on splash.bin in Windows Explorer and click on Properties to get the exact file size in bytes. Just look at the Size, not the Size On Disk. Take a look at Figure 9.3. For reference, I noted that the file length is 11,568 bytes.
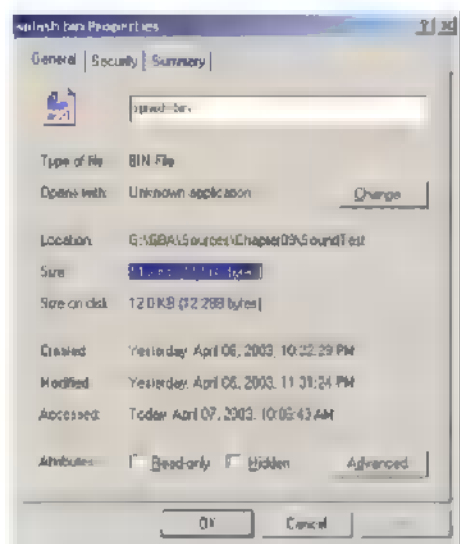
Figure 9.3

The splash.bin file is 11,568 bytes in length, which is the exact length of the sound sample needed for the SoundTest program.

The SoundTest Header File

If you have successfully converted the splash.wav file (or if you merely looked in the SoundTest folder and found that it has already been converted!), then you are ready for the source code to SoundTest. This program is fairly short, so I recommend that you type it into Visual HAM—this is akin to getting your hands greasy by working on an engine, as opposed to hiring someone else to repair your car. You learn a great deal by doing it yourself!

If you are writing this program yourself, you will need to create a new project in Visual HAM called SoundTest. Add a new file by selecting File, New, New File. Be sure to select the radio button Add To Project and click on the C Header icon on the left. I have called the file main.h, but you may call it whatever you like, as long as you include this file in the main.c file. The dialog box is shown in Figure 9.4.
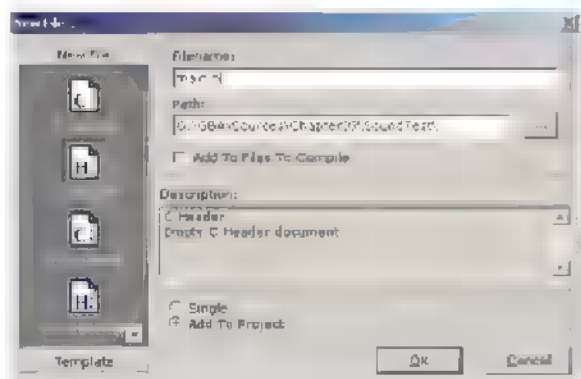
Figure 9.4

The New File dialog box in Visual HAM.

Now type the following code into the new main.h file. This code includes all the definitions needed to program the sound system on the GBA, including access to the DMA, timers, and interrupts needed to control sample playback.

```
///////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// SoundTest Project
// main.h header file
///////////////////////////////////////////////

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;

//define some video registers/values
#define REG_DISPCNT *(u32*)0x4000000
#define BG2_ENABLE 0x400
#define SetMode(mode) REG_DISPCNT = (mode)

//define some interrupt registers
#define REG_IME          *(u16*)0x4000208
#define REG_IE           *(u16*)0x4000200
#define REG_IF           *(u16*)0x4000202
#define REG_INTERRUPT    *(u32*)0x3007FFC
#define REG_DISPSTAT     *(u16*)0x4000004
#define INT_VBLANK       0x0001

//define some timer and DMA registers/values
#define REG_TM0D         *(volatile u16*)0x4000100
#define REG_TM0CNT       *(volatile u16*)0x4000102
#define REG_DMA1SAD      *(volatile u32*)0x40000BC
#define REG_DMA1DAD      *(volatile u32*)0x40000C0
#define REG_DMA1CNT_H    *(volatile u16*)0x40000C6
#define TIMER_ENABLE     0x80
#define DMA_DEST_FIXED   64
#define DMA_REPEAT       512
#define DMA_32           1024
#define DMA_ENABLE       32768
#define DMA_TIMING_SYNC_TO_DISPLAY  4096 | 8192

//define some sound hardware registers/values
```

```
#define REG_SGCNT0_H *(volatile u16*)0x4000082
#define REG_SGCNT1 *(volatile u16*)0x4000084
#define DSOUND_A_RIGHT_CHANNEL  256
#define DSOUND_A_LEFT_CHANNEL   512
#define DSOUND_A_FIFO_RESET     2048
#define SOUND_MASTER_ENABLE     128
```

The SoundTest Source File

Now for the main source code file of SoundTest. This code should be typed into the main.c file (replacing the default code added by Visual HAM when the project was created).

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// SoundTest Project
// main.c source code file
//////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"
#include "splash.c"

//global variables
u16 len = 0;

//////////////////////////////////////////////////
// Function: MyHandler
// Custom interrupt callback function
//////////////////////////////////////////////////
void MyHandler(void)
{
    //disable/store interrupts
    REG_IME = 0x00;
    u16 Int_Flag = REG_IF;

    //look for vertical refresh
    if((REG_IF & INT_VBLANK) == INT_VBLANK)
    {
        if (!len--)
```

```
            {
                //stop playback; disable the timer and DMA
                REG_TM0CNT = 0;
                REG_DMA1CNT_H = 0;
            }
        }

    //restore/enable interrupts
    REG_IF = Int_Flag;
    REG_IME = 0x01;
}

////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////////////
int main(void)
{
    u16 samplerate = 8000;
    u16 samplelen = 11568;
    u16 samples;

    SetMode(3 | BG2_ENABLE);

    //create custom interrupt handler for vblank (chapter 8)
    REG_IME = 0x00;
    REG_INTERRUPT = (u32)MyHandler;
    REG_IE |= INT_VBLANK;
    REG_DISPSTAT |= 0x08;
    REG_IME = 0x01;

    //output to both channels and reset the FIFO
    REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
        DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

    //enable all sound
    REG_SGCNT1 = SOUND_MASTER_ENABLE;

    //DMA1 source address
    REG_DMA1SAD = (u32)splash;

    //DMA1 destination address
```

```
        REG_DMA1DAD = 0x40000A0;

        //write 32 bits into destination every vblank
        REG_DMA1CNT_H =  DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
            DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

        //set the sample rate
        samples = 16777216 / samplerate;
        REG_TMOD = 65536 - samples;

        //determine length of playback in vblanks
        len = samplelen / samples * 15.57;

        //enable the timer
        REG_TM0CNT = TIMER_ENABLE;

        //run forever
        while(1);
        return 0;
}
```

Now that you have managed to get a sound to play through VisualBoyAdvance, I encourage you to create your own wave file and try to get it to play in the SoundTest program. The experience will be a valuable lesson in the process of converting a sound file, something you will end up doing often while working on a real game. Just remember, if you get any errors while trying to convert a wave file, you'll need to load it into a sound-editing program to downsample it to 8 bits, and you may also need to convert the wave to PCM. Make sure you are able to do this before moving on in the chapter.

The PlaySamples Program

The PlaySamples program is an interesting program that demonstrates how to handle multiple sounds on the GBA. While I would love to get into mixing, as I mentioned before, it is too difficult of a subject to cover here. Even if I were to develop a sound mixer with you in this chapter, it would not be optimized. There are prebuilt sound libraries for the GBA, most of which are written entirely in ARM7 assembly language, that are extremely efficient. Several open source and freeware libraries are available, as are professional ones like Krawall. Again, you can select a library that is suitable for your needs by perusing the Web sites listed in Appendix B.

The PlaySamples program is shown in Figure 9.5. This program demonstrates not only how to handle multiple sounds but also how to keep track of the current position within the sample as it is being played. The numbers shown in the figure are not bytes but rather are the number of samples played per vblank period.
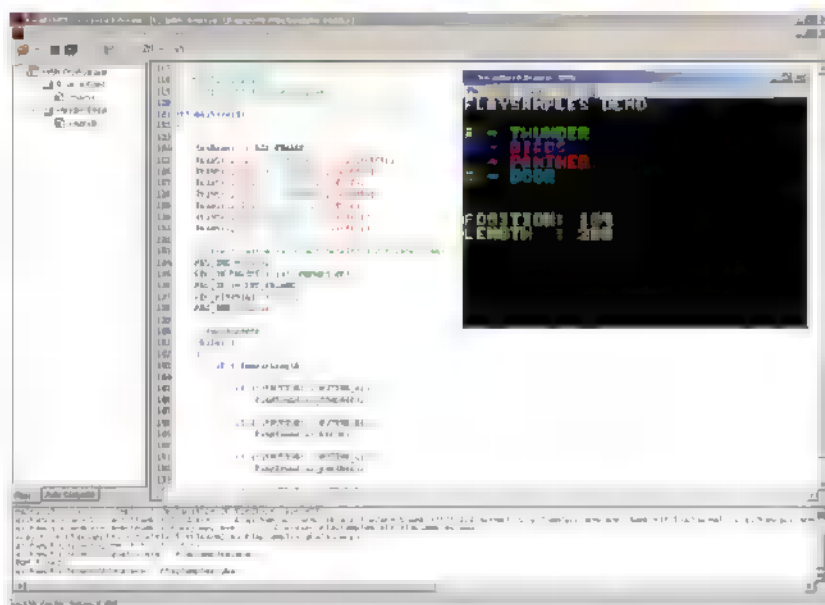
Figure 9.5

The PlaySamples program demonstrates how to keep track of several sounds in a program, as well as how to track the current playback position.

Tracking Sample Playback

The key to keeping track of the current playback position are two global variables, <u>Sample-Length</u> and <u>SamplePosition</u>. I am one of the first programmers to worry about using global variables in this manner, but as I have mentioned several times in the past, it is sometimes better to go with the brute force approach in console development. While SamplePosition is just set to 0, SampleLength is a bit more than just the byte count. It is actually the number of sample groups processed by the DAC at a timed interval specified by ■ timer. The calculation I used, which compensates for the CPU cycles per second, works out to two lines of code:

```
samples = 16777216 / samplerate;
SampleLength = samplelength / samples * 15.57;
```

The 15.57 simply compensates for the timer and would have been better without the decimal, but this is just setup code, so speed isn't critical.

The PlaySound Function

To facilitate the handling of multiple sound samples, I have converted the playback code from SoundTest into a reusable PlaySound function. Here is the complete function (which should look familiar to you after typing in the SoundTest program):

```
void PlaySound(sound *theSound)
{
    u16 samples;
```

```
    //output to both channels and reset the FIFO
    REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
    DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

    //enable all sound
    REG_SGCNT1 = SOUND_MASTER_ENABLE;

    //DMA1 source address
    REG_DMA1SAD = (u32)theSound->pBuffer;

    //DMA1 destination address
    REG_DMA1DAD = 0x40000A0;

    //write 32 bits into destination every vblank
    REG_DMA1CNT_H =  DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
        DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

    //set the sample rate
    samples = 16777216 / theSound->samplerate; //2097
    REG_TM0D = 65536 - samples;

    //keep track of the playback position and length
    SampleLength = theSound->length / samples * 15.57;
    SamplePosition = 0;

    //enable the timer
    REG_TM0CNT = TIMER_ENABLE;
}
```

Keeping Track of Sounds

You might have noticed that the PlaySound function had a sound parameter instead of a void* pointer to a sound buffer. The sound struct helps to keep track of samples used in the program, so there aren't just a bunch of arrays or global variables (or at least, there are as few as possible). Here is what the sound struct looks like:

```
typedef struct tagSound
{
    void *pBuffer;
    u16 samplerate;
    u32 length;
}sound;
```

Now I'm skipping over the #include statements that load the actual sounds into the program. The sample arrays used in the PlaySamples program are called panther, thunder, door, and birds. Here is how I created the structs to keep track of these sounds. Note that each initialization also includes the sample's rate and length (which you must specify yourself, since the bin2c program didn't provide these values).

```
sound s_panther = {&panther, 8000, 16288};

sound s_thunder = {&thunder, 8000, 37952};

sound s_door = {&door, 8000, 16752};

sound s_birds = {&birds, 8000, 29280};
```

Simply calling PlaySound with one of these sound variables (s_panther, s_thunder, s_door, or s_birds) will start playback of that particular sample. The sample length value helps to determine when the sound output should be halted; this is done inside the interrupt handler for vblank. Without listing the entire interrupt callback function, here's the key code that takes care of shutting down the sound output when playback has reached the end of the sample:

```
SamplePosition++;
if (SamplePosition > SampleLength)
{
    REG_TM0CNT = 0;
    REG_DMA1CNT_H = 0;
    SampleLength = 0;
}
```

Each time through the vblank interrupt, a check is made to determine if SamplePosition is greater than SampleLength. The sound is actually halted by turning off the timer and also the DMA controller, both of which are responsible for providing new bytes to the DAC. Obviously, SamplePosition and SampleLength are globals, so this code can handle just one sample at a time. There is no means to stop a sample during playback and then resume, but if you wanted to just stop playback of a sample—for instance, to play a different sample—then you could set both of these registers to 0 in a generic StopSound function. I elected to just set the values inside the interrupt, but a StopSound function would be useful in an actual game.

The PlaySamples Header File

Now that the theory is behind you, are you ready to get started on the source code for the PlaySamples program? It really is a simple program now that PlaySound has taken care of the details of setting up the sound registers and so on. So as in most cases, once the nitty-gritty is stuffed away in a reusable function, you can get down to business with the core program.

Okay, let's fire up Visual HAM and create a new project called PlaySamples. This program requires the font.h file, which you may copy from an earlier project, such as the Framerate program in the previous chapter. Add a new file to the PlaySamples project called main.h, and type in the following code:

```
/////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// PlaySamples Project
// main.h header file
/////////////////////////////////////////////

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

#include "font.h"
#include <stdlib.h>
#include <string.h>

//function prototypes
void Print(int left, int top, char *str, unsigned short color);
void DrawChar(int left, int top, char letter, unsigned short color);
void DrawPixel3(int x, int y, unsigned short color);

//define some video registers/values
unsigned short* videoBuffer = (unsigned short*)0x6000000;
#define REG_DISPCNT *(u32*)0x4000000
#define BG2_ENABLE 0x400
#define SetMode(mode) REG_DISPCNT = (mode)

//define some interrupt registers
#define REG_IME        *(u16*)0x4000208
#define REG_IE         *(u16*)0x4000200
#define REG_IF         *(u16*)0x4000202
#define REG_INTERRUPT  *(u32*)0x3007FFC
#define REG_DISPSTAT   *(u16*)0x4000004
#define INT_VBLANK     0x0001
```

```
//define some timer and DMA registers/values
#define REG_TM0D          *(volatile u16*)0x4000100
#define REG_TM0CNT        *(volatile u16*)0x4000102
#define REG_DMA1SAD       *(volatile u32*)0x40000BC
#define REG_DMA1DAD       *(volatile u32*)0x40000C0
#define REG_DMA1CNT_H     *(volatile u16*)0x40000C6
#define TIMER_ENABLE       0x80
#define DMA_DEST_FIXED     64
#define DMA_REPEAT         512
#define DMA_32             1024
#define DMA_ENABLE         32768
#define DMA_TIMING_SYNC_TO_DISPLAY  4096 | 8192

//define some sound hardware registers/values
#define REG_SGCNT0_H *(volatile u16*)0x4000082
#define REG_SGCNT1 *(volatile u16*)0x4000084
#define DSOUND_A_RIGHT_CHANNEL   256
#define DSOUND_A_LEFT_CHANNEL    512
#define DSOUND_A_FIFO_RESET      2048
#define SOUND_MASTER_ENABLE      128

//define button hardware register/values
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_R 256
#define BUTTON_L 512

/////////////////////////////////////////////
// Function: Print
// Prints a string using the hard-coded font
/////////////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color);
```

```
        pos += 8;
    }
}


/////////////////////////////////////////////////
// Function: DrawChar
// Draws a character one pixel at a time
/////////////////////////////////////////////////
void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            // grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            // if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
        }
}


/////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}


/////////////////////////////////////////////////////////
// Function: DrawBox3
// Draws a filled box
/////////////////////////////////////////////////////////
void DrawBox3(int left, int top, int right, int bottom,
    unsigned short color)
{
```

```
        int x, y;

        for(y = top; y < bottom; y++)
            for(x = left; x < right; x++)
                DrawPixel3(x, y, color);
}
```

The PlaySamples Source File

The main source code for the PlaySamples program should be typed into the main.c file
(and as usual, be sure to first delete the skeleton code added to the file by Visual HAM).
Now here is the main code for the program:

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// PlaySamples Project
// main.c source code file
//////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"
#include "panther.c"
#include "thunder.c"
#include "door.c"
#include "birds.c"

//create a struct to keep track of sound data
typedef struct tagSound
{
    void *pBuffer;
    u16 samplerate;
    u32 length;
}sound;

//create variables that describe the sounds
sound s_panther = {&panther, 8000, 16288};
sound s_thunder = {&thunder, 8000, 37952};
sound s_door = {&door, 8000, 16752};
```

```
sound s_birds = {&birds, 8000, 29280};

//global variables
u16 SamplePosition = 0;
u16 SampleLength = 0;
char temp[20];

//////////////////////////////////////////////
// Function: PlaySound
// Plays a sound sample using DMA
//////////////////////////////////////////////
void PlaySound(sound *theSound)
{
    u16 samples;

    //output to both channels and reset the FIFO
    REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
        DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

    //enable all sound
    REG_SGCNT1 = SOUND_MASTER_ENABLE;

    //DMA1 source address
    REG_DMA1SAD = (u32)theSound->pBuffer;

    //DMA1 destination address
    REG_DMA1DAD = 0x40000A0;

    //write 32 bits into destination every vblank
    REG_DMA1CNT_H =  DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
        DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

    //set the sample rate
    samples = 16777216 / theSound->samplerate; //2097
    REG_TM0D = 65536 - samples;

    //keep track of the playback position and length
    SampleLength = theSound->length / samples * 15.57;
    SamplePosition = 0;

    //enable the timer
    REG_TM0CNT = TIMER_ENABLE;
```

```
    }

    //////////////////////////////////////////////////
    // Function: MyHandler
    // Custom interrupt callback function
    //////////////////////////////////////////////////
    void MyHandler(void)
    {
        u16 Int_Flag;

        //disable interrupts
        REG_IME = 0x00;

        //backup the interrupt flags
        Int_Flag = REG_IF;

        //look for vertical refresh
        if((REG_IF & INT_VBLANK) == INT_VBLANK)
        {
            //is a sample currently playing?
            if (SampleLength)
            {
                //display the current playback position
                DrawBox3(80, 80, 120, 100, 0x0000);
                sprintf(temp, "%i", SamplePosition);
                Print(80, 80, temp, 0xDFFD);
                sprintf(temp, "%i", SampleLength);
                Print(80, 90, temp, 0xDFFD);

                //increment the position, check if complete
                SamplePosition++;
                if (SamplePosition > SampleLength)
                {
                    //stop playback: disable the timer and DMA
                    REG_TM0CNT = 0;
                    REG_DMA1CNT_H = 0;
                    //reset length
                    SampleLength = 0;
                }
            }
```

```
        }

        //restore the interrupt flags
        REG_IF = Int_Flag;

        //enable interrupts
        REG_IME = 0x01;
}

/////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////
int main(void)
{
    SetMode(3 | BG2_ENABLE);
    Print(0, 0, "PLAYSAMPLES DEMO", 0xFFFF);
    Print(0, 20, "A - THUNDER", 0x0FF0);
    Print(0, 30, "B - BIRDS", 0xF00F);
    Print(0, 40, "L - PANTHER", 0x00FF);
    Print(0, 50, "R - DOOR", 0xFF00);
    Print(0, 80, "POSITION: ", 0xCFFC);
    Print(0, 90, "LENGTH  : ", 0xCFFC);

    //create custom interrupt handler for vblank (chapter 8)
    REG_IME = 0x00;
    REG_INTERRUPT = (u32)MyHandler;
    REG_IE |= INT_VBLANK;
    REG_DISPSTAT |= 0x08;
    REG_IME = 0x01;

    //run forever
    while(1)
    {
        if (!SampleLength)
        {
            if (!(*BUTTONS & BUTTON_A))
                PlaySound(&s_thunder);

            if (!(*BUTTONS & BUTTON_B))
                PlaySound(&s_birds);
```

```
                    if (!(*BUTTONS & BUTTON_L))

                          PlaySound(&s_panther);

                    if (!(*BUTTONS & BUTTON_R))

                          PlaySound(&s_door);

               }

          }

     return 0;

}
```

If all goes well, you should now have a simple but useful sound engine for your next game project. Sound effects are at least as important as the graphics in a game, so don't put sound on the side while working on the "more important" aspects of your next game. A well-designed game uses sound to greatly enhance the gaming experience.

Summary

This chapter has been an overview of the sound hardware on the GBA. You learned about the Direct Sound A and Direct Sound B channels and how to create, convert, and play samples. This chapter provided a simple demonstration of playing ■ single sound, followed by ■ more useful program that was able to play one of several sound effects based on button presses. There is obviously more to sound programming than has been covered in this single chapter, but you now have enough information to add sound support to your GBA programs. For more advanced sound capabilities, including the ability to play modules as music tracks in addition to sound mixing, I recommended going with a sound library such as Krawall, since it is rare even among commercial GBA developers to write a custom sound library when excellent prebuilt solutions are already available for a small licensing fee.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter. The solution to each challenge is provided on the CD-ROM inside the folder for this chapter.

Challenge 1: The SoundTest program is a simple and easy way to test converted wave files. See if you can convert your own wave files with wav2gba and bin2c, as explained in this chapter, to gain some experience converting and playing sound files.

Challenge 2: The PlaySamples program displays the playback position in sample blocks per vblank. Modify the program so it shows both the position and length of the sample in actual bytes.

Challenge 3: The PlaySamples program supports just four sounds, using the A, B, L, and R buttons. Enhance the program by adding more sounds and make use of the other buttons: Up, Down, Left, Right, Start, and Select.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in Appendix D.

1. How many *total* sound channels are built into the GBA sound system?

    A. 2

    B. 4

    C. 6

    D. 8

2. True or False: The GBA sound system supports stereo sound.

    A. True

    B. False

3. What are the two digital sound channels called?

    A. Frequency and Modulation

    B. Digital Sound 1 and Digital Sound 2

    C. Direct Sound A and Direct Sound B

    D. FM Synthesis and Wave Table

4. What utility program is used in this chapter to convert a wave file?

    A. wav2c

    B. wav2bin

    C. bin2long

    D. wav2gba

5. What sampling resolution is supported by the GBA's sound system?

    A. 8-bit

    B. 12-bit

    C. 16-bit

    D. 24-bit

6. What does it mean if a sound sample has a frequency of 44.1 kHz?

    A. The sample will be played back quickly.

    B. The sound has been undersampled.

C. The sample was recorded from a radio station.

D. The sample contains 44,100 samples per second.

7. What is the name of the direct sound control register at memory address 0x04000082?

    A. REG_SNDCNT

    B. REG_SOUND_CNT_L

    C. REG_SOUND_CNT_H

    D. REG_DS_CNT

8. How many CPU cycles does the GBA execute per second?

    A. 32,768

    B. 16,777,216

    C. 1,024

    D. 65,535

9. What wave file format does the GBA sound system support exclusively?

    A. PCM

    B. A/mu-Law Wave

    C. ACM Waveform

    D. DVI/IMA ADPCM

10. What is the name of the sound mixing and ProTracker music playback library mentioned in this chapter?

    A. Tidal Wave

    B. Cool Tunes

    C. Kurzweil

    D. Krawall

Chapter 10

Interfacing with the Buttons

onsole video game machines such as Game Boy Advance differ greatly from PCs in the realm of input. On a console, there is basically just the stock controller, with the occasional driving wheel or even more esoteric foot pad (for running or dancing games). The PC, however, uses a keyboard and mouse for input, with support for a whole slew of input devices, such as joysticks, gamepads, flight yokes with foot pedals, driving wheels. . . the list goes on. This chapter explains how to program the Game Boy Advance for detecting button input—the sole means of input on this system. You have already seen a sample of how to program the buttons on the GBA, from the ButtonTest program in Chapter 4. This chapter goes much further and explains every aspect of the button hardware on the GBA.

Here are the main topics of this chapter:

- The button architecture
- Detecting button input
- Creating a button handler

The Button Architecture

The Game Boy Advance features 10 buttons that may be used for input by any game. While there are standards by which GBA games have come to follow, such as using the Start and Select buttons exclusively for management-type functions (starting the game, pausing the game, bringing up a menu, and so on), the game designer or programmer is not so limited from ■ programming point of view, as all of the buttons are treated equally in code. What you do with the buttons is entirely up to you (within the limits of the game design, of course). Figure 10.1 shows the placement of the buttons on a GBA.

Figure 10.1

The Game Boy Advance features 10 distinct gameplay buttons.

Detecting Button Input

Like all other aspects of the GBA, the status of the buttons is placed in a specific location in memory that must be polled by your program. The memory address for the button status is at 0x04000130. This is simply a number that you must define as a constant, like all other memory address constants in the GBA. If you want to memorize them all, be my guest! But it is always easier to write something down rather than memorize it, particularly when writing games. This is especially true if you get into cross-platform development. For instance, I do a lot of Pocket PC programming, and the Pocket PC devices use primarily the StrongARM processor, which is a close relative of the ARM7 processor used in the GBA! Of course, after using something for a long while, you come to recognize it and even begin to commit many things to memory as a matter of course.

The button status value at 0x04000130 is a 32-bit number. By the way, almost all memory locations are 32-bit simply because the GBA is a 32-bit machine. Since there are 10 buttons for input, and typically only 8 are used for gameplay, it doesn't take much to identify ■ button press on the part of the hardware—it simply sets a bit on or off based on the status of the button. Now, how about creating a pointer to this memory location so it's easy to check for button input in a game? On the GBA, a 32-bit number is called an *int* or *unsigned*

int. (Remember that a short is 16 bits.) Therefore, to create a pointer to this memory address, you would do this:

```
unsigned int *BUTTONS = (unsigned int *)0x04000130;
```

Throughout this book, I have assumed that you already know C, but just for reference, the (unsigned int *) is a typecast that ensures the *BUTTONS pointer grabs the whole 32 bits of the memory address. As a precaution, this pointer should be declared with the *volatile* keyword, to tell the compiler that it is a memory address that is changed by another process (not by the program itself):

```
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
```

There you have it—a new pointer to the memory address where button values are stored by the GBA.

Searching for Buttons

Now, suppose you don't know what the various button values are and just want to check to see if any button has been pressed at all. For instance, suppose you are displaying some sort of video on the screen as an introduction to your game and want to just check for any button press to halt the video and go straight to the start screen of the game. You might just check to see of the memory address pointed to by BUTTONS contains anything other than 0. Let's write a short code snippet to do just that:

```
while(1)
{
    if(*BUTTONS)
        // yes, a button was pressed!
    else
        // nothing yet
}
```

All this code snippet does is check for any value other than 0 (which is considered a false Boolean value by the if statement, as you well know). The only problem with this code is that it always returns a true value, whether a button is being pressed or not! Why is that, do you suppose? I scratched my head about it myself for a while, until I came up with an idea. How about writing a program that performs a loop and scans for possible numbers and then run the program, press a button, and see what number comes up. That's not a bad idea, actually. Let's write a short program to do that! I'll use the Print function and the usual font.h file to display messages on the screen.

The ScanButtons Program

Now, let's create the ScanButtons project. Fire up Visual HAM and create a new project by opening the File menu and selecting New, New Project. As I've gone over this many times already, I'll assume you are proficient with Visual HAM and no longer need a tutorial on creating the project and so on. Name the project ScanButtons, and then add the font.h file to the project. In case you have forgotten, you can add a file by right-clicking on the project workspace, either the Source Files or Header Files section, selecting Add File(s) to Folder, and then searching for the font.h file from \Sources\Chapter07. I prefer to copy the font.h file to my new project folder, which makes it easier to add the file to the project.

The point is that you don't have to type in the font code again, so whatever works for you, go with it.

The #include <stdio.h> tells the compiler to include the standard ANSI C input/output library, which includes the uber-cool sprintf function.

Visual HAM automatically adds the main.c file to the project, and this file is where you should type in the source code for the program. If you want, you may copy the DrawPixel3, Print, PrintT, and DrawChar functions from earlier programs rather than typing them in again. I could put these common functions into a separate code file, but that always leaves room for error for those who are not adept at managing projects in C, including header files and so on. It's just easier to include short and common functions again.

```
///////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ScanButtons Project
// main.c source code file
///////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include <stdio.h>
#include "font.h"

//define boolean
#define bool short
#define true 1
```

```
#define false 0


//define some useful colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D


//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000


//background 2
#define BG2_ENABLE 0x400


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;


//declare some function prototypes
void SetMode(int mode);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);
void PrintT(int, int, char *, unsigned short);
void DrawChar(int, int, char, unsigned short, bool);


//////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
```

```
//////////////////////////////////////////////////////////////
int main()
{
    char str[20];

    SetMode(3);
    Print(1, 1, "SCANBUTTONS PROGRAM", WHITE);
    Print(1, 30, "SCANNING...", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        if (*BUTTONS)
        {
            sprintf(str, "BUTTON CODE = %i    ", (unsigned int)*BUTTONS);
            Print(10, 40, str, BLUE);
        }
    }

    return 0;
}


void SetMode(int mode)
{
    REG_DISPCNT = (mode | BG2_ENABLE);
}


//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draw a pixel on the mode 3 video buffer
//////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
```

```
        videoBuffer[y * 240 + x] = c;
}


///////////////////////////////////////////////////////////
// Function: Print
// Prints a string using a hard-coded font, must be all caps
///////////////////////////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color, false);
        pos += 8;
    }
}


///////////////////////////////////////////////////////////
// Function: PrintT
// Prints a string with transparency
///////////////////////////////////////////////////////////
void PrintT(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color, true);
        pos += 8;
    }
}


///////////////////////////////////////////////////////////
// Function: DrawChar
// Draws a single character from a hard-coded font
```

```
//////////////////////////////////////////////////////////
void DrawChar(int left, int top, char letter,
              unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}
```

When you finish typing in the code for the ScanButtons program, hit F7 to run the program. If there are no typos in the program, you should see output that looks like Figure 10.2. A glance at the program shows a problem right away. The default value without pressing any buttons is 1,023.

Making Sense of Button Values

What this means is that all the bits are set to 1 by default and are individually set to 0 when a button is pressed. The problem is that the GBA sets status bits based on position rather than just storing an arbitrary value! That's right. So while you might expect to see buttons Start=1, Select=2, and perhaps A=3, B=4, and so on, that is not what is really going on. If you were to just check for a specific value, you would be able to detect only one button press at a time, which is definitely not the way this is supposed to work. Instead of looking at *BUTTONS for a specific value, we need to look at the bitmask. The GBA sets an incremental bit based on the button press, and it looks like Figure 10.3.
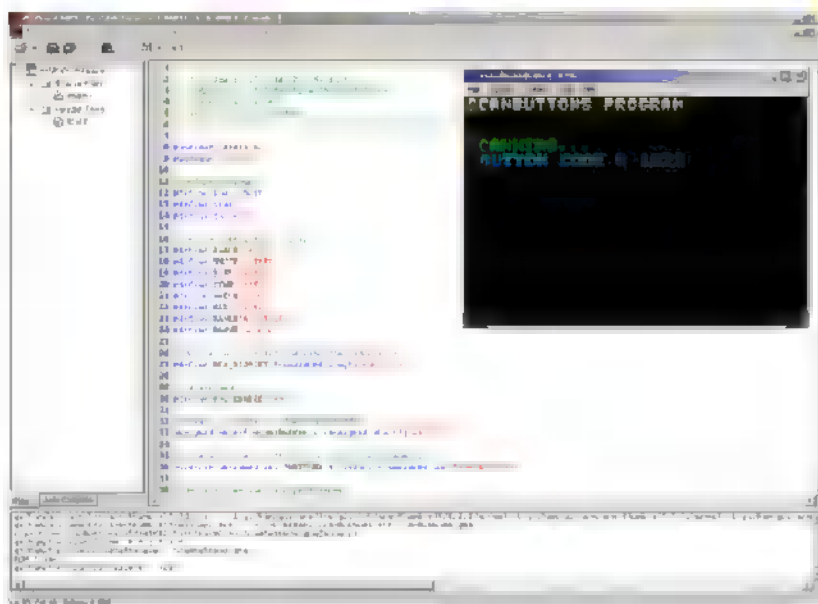
Figure 10.2

The ScanButtons program displays the button input status value.

Memory Address 0x04000130

BITS	VALUES	BUTTONS
0	1	A
1	2	B
2	4	SELECT
3	8	START
4	16	RIGHT
5	32	LEFT
6	64	UP
7	128	DOWN
8	256	R
9	512	L
10		
·		NOT USED
·		
31		

Figure 10.3

The bit values of the button status memory location.

This calls for a little change to the program, because the button values will need to be AND'ed to this 1,023 in order to determine which button was pressed (and it's necessary in order to detect multiple button presses).

Correctly Identifying Buttons

Let's modify the program to fix the problem, so that it only reports a value if one or more buttons are actually pressed. If you want to just load this project off the CD-ROM instead of making the changes by hand, the corrected version is on the CD-ROM under \Sources\Chapter10\ScanButtons2. The only change needed involves the while(1) loop. Here is the new version:

```
// continuous loop
while(1)
{
    //check for button presses
    for (n=1; n < 1000; n++)
    {
        if (!((*BUTTONS) & n))
        {
            sprintf(str, "BUTTON CODE = %i   ", n);
            Print(10, 40, str, BLUE);
            break;
        }
    }
}
```

The first thing you'll likely have noticed is the for loop that goes from 1 to 1,000. That is just an arbitrary number that will accommodate all 10 buttons, because I don't know immediately which bits the GBA uses for each button. I know that it will likely only go up to 512, but this is knowledge gained after the fact. From a fact-seeking point of view, one might not necessarily know this in advance (no pun intended). Now, hurry up and run the program! The output is shown in Figure 10.4.

I was pressing the UP button in the screen shot shown, which indicates a value of 64. Using this little button-scanning program, you can come up with a list of values for each button! Here is the table I came up with after noting the value of each button reported by ScanButtons2 (see Table 10.1).
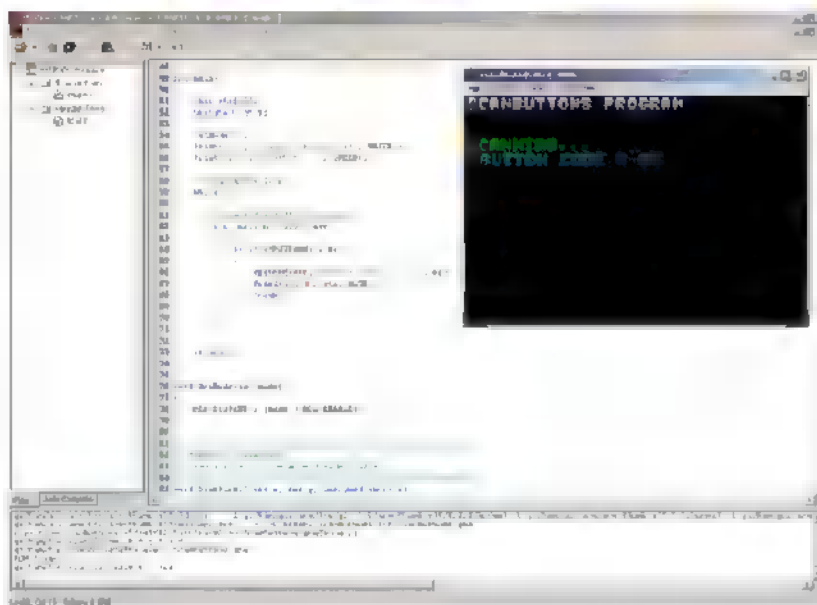
Figure 10.4

The modified ScanButtons2 program displays a value only if a button is pressed.

This is a really good lesson in dealing with low-level hardware interfaces. Being able to figure out something on your own is worth a thousand tutorials on the Web! The reasoning being that these concepts apply to other consoles and computer hardware, not just the GBA.

Table 10.1 Button Input Values

Button	Value
A	1
B	2
SELECT	4
START	8
RIGHT	16
LEFT	32
UP	64
DOWN	128
R	256
L	512

Displaying Button Scan Codes

As you have likely spent some time writing programs for your PC, you probably know all about keyboard scan codes. The GBA's buttons have codes too, although they are not exactly like the ASCII codes on a PC. If you have done any low-level keyboard programming though, where each key is numbered in sequence, it does start to resemble the buttons on ▪ GBA a little more.

Now let's modify the program again so that it shows the name of the button being pressed. Again, this just involves changing the main while loop—but I've used ▪ new function called strcpy in order to display the button name, so you will need to add another #include statement to the top of the program as follows:

```
#include <string.h>
```

You will also need to add a new variable to the program, called name:

```
char name[10];
```

I have saved this modified version as ScanButtons3. Here is the new version of the main functon:

```
int main()
{
    char str[20];
    unsigned int n;
    char name[10];

    SetMode(3);
    Print(1, 1, "SCANBUTTONS PROGRAM", WHITE);
    Print(1, 30, "SCANNING...", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        for (n=1; n < 1000; n++)
        {
            if (!((*BUTTONS) ▪ n))
            {
                sprintf(str, "BUTTON CODE = %i   ", n);
```

```
                    Print(10, 40, str, BLUE);


                    //figure out the button name
                    switch(n)
                    {
                         case 1: strcpy(name, "A"); break;
                         case 2: strcpy(name, "B"); break;
                         case 4: strcpy(name, "SELECT"); break;
                         case 8: strcpy(name, "START"); break;
                         case 16: strcpy(name, "RIGHT"); break;
                         case 32: strcpy(name, "LEFT"); break;
                         case 64: strcpy(name, "UP"); break;
                         case 128: strcpy(name, "DOWN"); break;
                         case 256: strcpy(name, "R"); break;
                         case 512: strcpy(name, "L"); break;
                         default: strcpy(name, ""); break;
                    }
                    sprintf(str, "BUTTON NAME = %s     ", name);
                    Print(10, 50, str, RED);
                    break;
               } //if
          } //for
     } //while

} //main
```
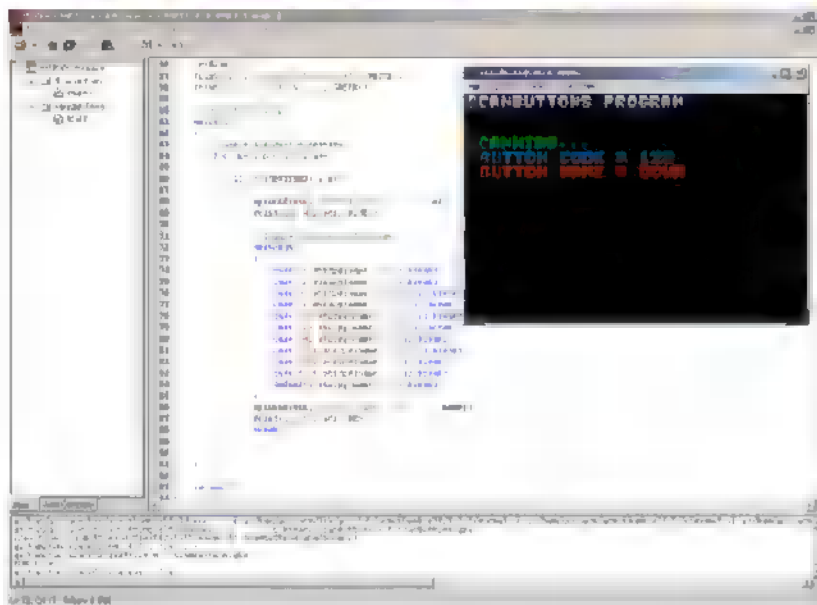
The output from the ScanButtons3 program is shown in Figure 10.5.

Figure 10.5

The modified ScanButtons3 program now displays the button name.

Getting the Hang of Button Input

Now that you know the value of each button, we can put together a list of #define statements to make it easier to use the buttons in your GBA programs. Here is the list that I came up with:

```
#define BUTTON_A 1

#define BUTTON_B 2

#define BUTTON_SELECT 4

#define BUTTON_START 8

#define BUTTON_RIGHT 16

#define BUTTON_LEFT 32

#define BUTTON_UP 64

#define BUTTON_DOWN 128

#define BUTTON_R 256

#define BUTTON_L 512
```

This list of button constants, along with the *BUTTONS pointer, allows you to make robust use of button input in all future GBA programs. Since the code is so meager, it's not a big deal to copy the code from one project to another (in other words, there is no real need for a separate .lib). Some folks get all worked up about libraries of code and making everything reusable and so on, but sometimes it's just preferable to see the code directly. Once you have a large collection of GBA code that you use frequently, it is then a good idea to lib it up!

Creating a Button Handler

A button handler takes the actual button code and abstracts it to a second level in order to make the source code more intuitive. Since button input is a very low-level aspect of programming the GBA, it's helpful to move the actual memory reading code into a function and store the results of the buttons in an array. Of course, this is not at all necessary and only reflects my own coding style. If you prefer to put the button code directly in your main loop, feel free to do so. The main benefit for polling all the buttons at the same time is that you are more likely to lose a button if there is a lot of code between each poll. For instance, if your program detects the A button has been pressed, and does some processing based on that input, but then checks to look for a simultaneous B button press, that button may no longer register. By polling the state of all buttons in one quick interval, you may then use up the entire vblank period to handle the input of the buttons without worrying about losing a button.

For instance, if you use the A button to fire a weapon and the D-pad to move a ship on the screen, then your weapon firing code may likely take up so many cycles that you miss the D-pad input. Just think of it in logical terms. It makes more sense to segregate your main loop code into easily identifiable sections, having button input separate from graphics output, sound generation, and game logic. Once buttons are polled at the start of the game loop, it is then an easy matter later on in the loop to check the status of each button as needed.

Handling Multiple Buttons

To abstract the button handler from the low-level button code, an array is needed to store a simple Boolean true or false value (which equates to 1 or 0, respectively):

```
bool buttons[10];
```

Using the button array is simply a matter of polling all the buttons at once and storing the result of each poll in an element of the array as follows. Note that the assignment causes C to evaluate the expression for a true or false. The result is negated because the result of the AND operation is a 0 when the button is pressed, which is the opposite of what we want—a button press should equate to 1.

```
buttons[0] = !((*BUTTONS) & BUTTON_A);

buttons[1] = !((*BUTTONS) & BUTTON_B);

buttons[2] = !((*BUTTONS) & BUTTON_LEFT);

buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);

buttons[4] = !((*BUTTONS) & BUTTON_UP);

buttons[5] = !((*BUTTONS) & BUTTON_DOWN);

buttons[6] = !((*BUTTONS) & BUTTON_START);
```

```
buttons[7] = !((*BUTTONS) & BUTTON_SELECT);

buttons[8] = !((*BUTTONS) | BUTTON_L);

buttons[9] = !((*BUTTONS) & BUTTON_R);
```

Polling the buttons is a very fast process, so there really is no need to strip out buttons that you don't use, because this code could end up in a library eventually. If you plan to have a single code listing with this function in your main.c file, for instance, then you may want to comment out any buttons that you don't plan to use in the game. While the timing is negligible, every clock cycle does help. When it comes time to check for the button presses (presumably, that would be later on in the game loop), you can use a function such as the following to return a true or false value based on the button that is passed to the function:

```
bool Pressed(int button)

{

    switch(button)

    {

        case BUTTON_A: return buttons[0];

        case BUTTON_B: return buttons[1];

        case BUTTON_LEFT: return buttons[2];

        case BUTTON_RIGHT: return buttons[3];

        case BUTTON_UP: return buttons[4];

        case BUTTON_DOWN: return buttons[5];

        case BUTTON_START: return buttons[6];

        case BUTTON_SELECT: return buttons[7];

        case BUTTON_L: return buttons[8];

        case BUTTON_R: return buttons[9];

    }

}
```

The ButtonHandler Program

As you have seen, the button handler need not be complicated unless you want to detect button *releases* separately from button *presses*. Then it requires a little more thought and will most likely require another array to keep track of which buttons have been pressed. I have never needed to check for button releases, because when it comes to GBA coding, all that really matters is responding to a button press.

I have written a complete program to demonstrate how to write your own general-purpose button handler. The project is called ButtonHandler and is located on the CD-ROM under

\Source\Chapter10\ButtonHandler. As usual, this project requires the font.h file, so if you are creating this project from scratch and typing it in, be sure to add the font.h file to the folder for this project. Note that adding a file to the actual workspace in Visual HAM is not necessary—and actually, such files are not automatically compiled. Adding them to the file list simply makes it easier to edit the files in the project. The compiler only looks at header files that are included in a main source file with the #include directive. So it is sufficient to simply copy the font.h file to the project folder for each program that needs to display output. When you aren't using the built-in Hamlib functionality in your program, a font subsystem like this is essential, as you have seen over the last few chapters.

Now, here is the source code for the ButtonHandler project. Type it into the main.c file, and feel free to copy duplicate functions from other programs you have already typed in (such as DrawPixel3, Print, and so on).

```c
///////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ButtonHandler Project
// main.c source code file
///////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


#include "font.h"


//define boolean
#define bool short
#define true 1
#define false 0


//declare some function prototypes
void CheckButtons();
bool Pressed(int);
void SetMode(int);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);
void DrawChar(int, int, char, unsigned short, bool);
```

```
//define some colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D


//define the buttons
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512


//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000


//use background 2
#define BG2_ENABLE 0x400


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
```

```cpp
//keep track of the status of each button
bool buttons[10];


/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main()
{
    SetMode(3);
    Print(1, 1, "BUTTONHANDLER PROGRAM", WHITE);
    Print(1, 30, "PRESS A BUTTON:", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        CheckButtons();

        //display the status of each button
        if (Pressed(BUTTON_A))
            Print(10, 40, "A PRESSED", BLUE);
        else
            Print(10, 40, "        ", 0);
        if (Pressed(BUTTON_B))
            Print(10, 50, "B PRESSED", BLUE);
        else
            Print(10, 50, "        ", 0);
        if (Pressed(BUTTON_SELECT))
            Print(10, 60, "SELECT PRESSED", BLUE);
        else
            Print(10, 60, "              ", 0);
        if (Pressed(BUTTON_START))
```

```
                Print(10, 70, "START PRESSED", BLUE);

        else

                Print(10, 70, "              ", 0);

        if (Pressed(BUTTON_LEFT))

                Print(10, 80, "LEFT PRESSED", BLUE);

        else

                Print(10, 80, "            ", 0);

        if (Pressed(BUTTON_RIGHT))

                Print(10, 90, "RIGHT PRESSED", BLUE);

        else

                Print(10, 90, "             ", 0);

        if (Pressed(BUTTON_UP))

                Print(10, 100, "UP PRESSED", BLUE);

        else

                Print(10, 100, "          ", 0);

        if (Pressed(BUTTON_DOWN))

                Print(10, 110, "DOWN PRESSED", BLUE);

        else

                Print(10, 110, "            ", 0);

        if (Pressed(BUTTON_R))

                Print(10, 120, "R PRESSED", BLUE);

        else

                Print(10, 120, "         ", 0);

        if (Pressed(BUTTON_L))

                Print(10, 130, "L PRESSED", BLUE);

        else

                Print(10, 130, "         ", 0);

    }

    return 0;

}


//////////////////////////////////////////////////////////////

// Function: CheckButtons

// Polls the status of all the buttons
```

```
//////////////////////////////////////////////////////////////
void CheckButtons()
{
    //store the status of the buttons in an array
    buttons[0] = !((*BUTTONS) & BUTTON_A);

    buttons[1] = !((*BUTTONS) & BUTTON_B);

    buttons[2] = !((*BUTTONS) & BUTTON_LEFT);

    buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);

    buttons[4] = !((*BUTTONS) & BUTTON_UP);

    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);

    buttons[6] = !((*BUTTONS) & BUTTON_START);

    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);

    buttons[8] = !((*BUTTONS) & BUTTON_L);

    buttons[9] = !((*BUTTONS) & BUTTON_R);
}


//////////////////////////////////////////////////////////////
// Function: Pressed
// Returns the status of a button
//////////////////////////////////////////////////////////////
bool Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons[1];
        case BUTTON_LEFT: return buttons[2];
        case BUTTON_RIGHT: return buttons[3];
        case BUTTON_UP: return buttons[4];
        case BUTTON_DOWN: return buttons[5];
        case BUTTON_START: return buttons[6];
        case BUTTON_SELECT: return buttons[7];
        case BUTTON_L: return buttons[8];
        case BUTTON_R: return buttons[9];
```

```
        }

        return false;

    }


    ////////////////////////////////////////////////////////////
    // Function: SetMode
    // Changes the video mode
    ////////////////////////////////////////////////////////////
    void SetMode(int mode)

    {

        REG_DISPCNT = (mode | BG2_ENABLE);

    }


    ////////////////////////////////////////////////////////////
    // Function: DrawPixel3
    // Draw a pixel on the mode 3 video buffer
    ////////////////////////////////////////////////////////////
    void DrawPixel3(int x, int y, unsigned short c)

    {

        videoBuffer[y * 240 + x] = c;

    }


    ////////////////////////////////////////////////////////////
    // Function: Print
    // Prints a string using a hard-coded font, must be all caps
    ////////////////////////////////////////////////////////////
    void Print(int left, int top, char *str, unsigned short color)

    {

        int pos = 0;

        while (*str)

        {

            DrawChar(left + pos, top, *str++, color, false);

            pos += 8;

        }
```

```
}


/////////////////////////////////////////////////////////
// Function: DrawChar
// Draws a single character from a hard-coded font
/////////////////////////////////////////////////////////
void DrawChar(int left, int top, char letter,
              unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}
```

Now if you run the ButtonHandler program, you should see output that looks like the screen shot in Figure 10.6.
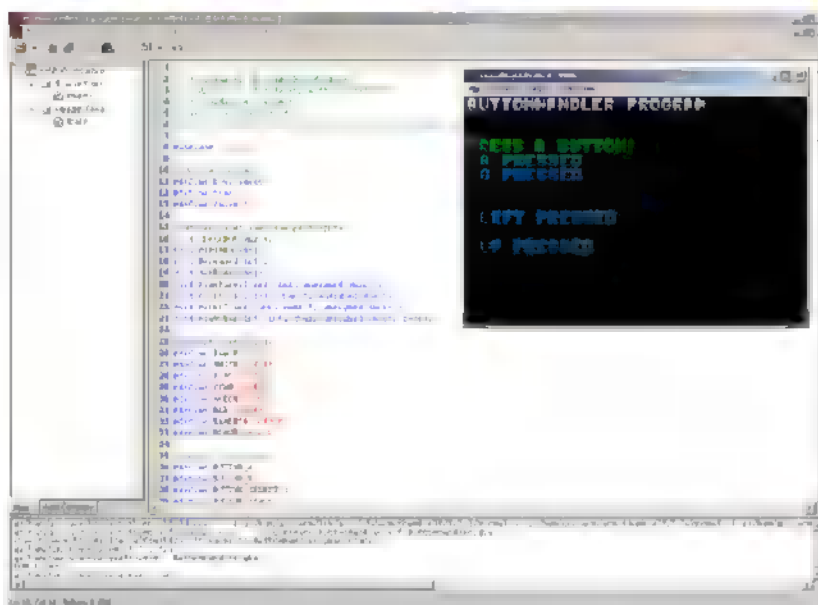
Figure 10.6

The ButtonHandler program shows how to write a reusable button handler.

Detecting Button Combos

Combos are a big aspect of many games, primarily fighting games like *Soul Calibur, Dead or Alive 3, Street Fighter* series, and so on. A combo is a series of button pushes in a specific order that gives the player's character some kind of power-up or super attack. For example, the infamous "sho'riuken!" dragon punch from *Street Fighter II*'s Ken character is almost infamous in videogame legend. Another popular combo for Ken (and Ryu) was the fireball, which required a combo of Down, Down+Right, Right, A to unleash a powerful uppercut (and the combo is reversed depending on the direction that the player is facing). What this means is that you must press D, DR, R, A in rapid succession to unleash the combo attack.

In a real game, timing is a critical issue, as you learned back in Chapter 8, "Using Interrupts and Timers." But to keep this example program short, I have cheated a little and just put sort of a hard-coded delay into the program to slow it down (interrupt and timer code is somewhat lengthy, as you'll recall). Button input occurs so quickly that it's impossible to detect combos without a delay; as soon as you touch a key, several input events fire off! The problem with this, when trying to detect a combo, is that the first button is registered several times on first press. So I'll just use a small loop to slow the program a bit. Following is a program called ComboTest, which demonstrates how to put combo support into your game.

In order to detect a combo, it is necessary to first identify how many buttons are being pressed and only start counting combinations if a single button is being pressed. To do this, I wrote a function called ButtonsPressed:

```
int ButtonsPressed()
{
```

```
        int n;
        int total = 0;
        for (n=0; n < 10; n++)
            total += buttons[n];
        return total;
}
```

Here's the source code for the complete ComboTest program. Just type this into the main.c
file for a new project, and be sure to copy the font.h file to the project folder as usual.
Again, there's some duplicated source code here, so feel free to copy and paste as needed.

```
/////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ComboTest Project
// main.c source code file
/////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include <stdio.h>
#include "font.h"

//define boolean
#define bool short
#define true 1
#define false 0

//declare some function prototypes
int ButtonsPressed();
void CheckButtons();
bool Pressed(int);
void SetMode(int);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);
```

```
void DrawChar(int, int, char, unsigned short, bool);


//define some colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D


//define the buttons
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512


//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000


//use background 2
#define BG2_ENABLE 0x400


//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;


//create pointer to the button interface in memory
```

```
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;


//keep track of the status of each button
bool buttons[10];


//////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////
int main()
{
    int counter = 0;
    char str[20];
    int delay;
    int combo[10] = {BUTTON_UP,BUTTON_UP,BUTTON_DOWN,BUTTON_DOWN,
        BUTTON_LEFT,BUTTON_RIGHT,BUTTON_LEFT,BUTTON_RIGHT,
        BUTTON_B,BUTTON_A};


    SetMode(3);
    Print(1, 1, "COMBOTEST PROGRAM", WHITE);
    Print(1, 30, "PRESS U,U,D,D,L,R,L,R,B,A:", GREEN);


    // continuous loop
    while(1)
    {
        //check for button presses
        CheckButtons();

        if (ButtonsPressed() == 1)
        {
            sprintf(str, "COUNTER = %i ", counter + 1);
            Print(10, 50, str, BLUE);

            if (Pressed(combo[counter]))
```

```
                {
                    counter++;

                    if (counter == 10)
                    {
                        Print(10, 70, "CHEAT MODE ACTIVATED!", RED);
                        counter = 0;
                    }

                    //slow down the program
                    delay = 500000;
                    while (delay--);
                }
                else
                    counter = 0;
            }
        }
        return 0;
}


/////////////////////////////////////////////////////////
// Function: ButtonsPressed
// Returns the number of buttons being pressed
/////////////////////////////////////////////////////////
int ButtonsPressed()
{
    int n;
    int total = 0;
    for (n=0; n < 10; n++)
        total += buttons[n];
    return total;
}


/////////////////////////////////////////////////////////
```

```cpp
// Function: CheckButtons
// Polls the status of all the buttons
///////////////////////////////////////////////////////
void CheckButtons()
{
    //store the status of the buttons in an array
    buttons[0] = !((*BUTTONS) & BUTTON_A);
    buttons[1] = !((*BUTTONS) & BUTTON_B);
    buttons[2] = !((*BUTTONS) & BUTTON_LEFT);
    buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);
    buttons[4] = !((*BUTTONS) & BUTTON_UP);
    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);
    buttons[6] = !((*BUTTONS) & BUTTON_START);
    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);
    buttons[8] = !((*BUTTONS) & BUTTON_L);
    buttons[9] = !((*BUTTONS) & BUTTON_R);
}


///////////////////////////////////////////////////////
// Function: Pressed
// Returns the status of a button
///////////////////////////////////////////////////////
bool Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons[1];
        case BUTTON_LEFT: return buttons[2];
        case BUTTON_RIGHT: return buttons[3];
        case BUTTON_UP: return buttons[4];
        case BUTTON_DOWN: return buttons[5];
        case BUTTON_START: return buttons[6];
        case BUTTON_SELECT: return buttons[7];
```

```
            case BUTTON_L: return buttons[8];

            case BUTTON_R: return buttons[9];

    }

    return false;

}


//////////////////////////////////////////////////////////

// Function: SetMode

// Changes the video mode

//////////////////////////////////////////////////////////

void SetMode(int mode)

{

    REG_DISPCNT = (mode | BG2_ENABLE);

}


//////////////////////////////////////////////////////////

// Function: DrawPixel3

// Draw a pixel on the mode 3 video buffer

//////////////////////////////////////////////////////////

void DrawPixel3(int x, int y, unsigned short c)

{

    videoBuffer[y * 240 + x] = c;

}


//////////////////////////////////////////////////////////

// Function: Print

// Prints a string using a hard-coded font, must be all caps

//////////////////////////////////////////////////////////

void Print(int left, int top, char *str, unsigned short color)

{

    int pos = 0;

    while (*str)

    {

        DrawChar(left + pos, top, *str++, color, false);
```

```
            pos += 8;
    }
}


/////////////////////////////////////////////////////////////////
// Function: DrawChar
// Draws a single character from a hard-coded font
/////////////////////////////////////////////////////////////////
void DrawChar(int left, int top, char letter, unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}
```

The output from the ComboTest program is shown in Figure 10.7. In case you were
wondering, the combo used in this program was inspired by the infamous NES version of
Contra: U-U-D-D-L-R-L-R-B-A-B-A. I think the game actually required you to hit Select or
Start at the end of the combo, which became known as the "Konami code" over the years.
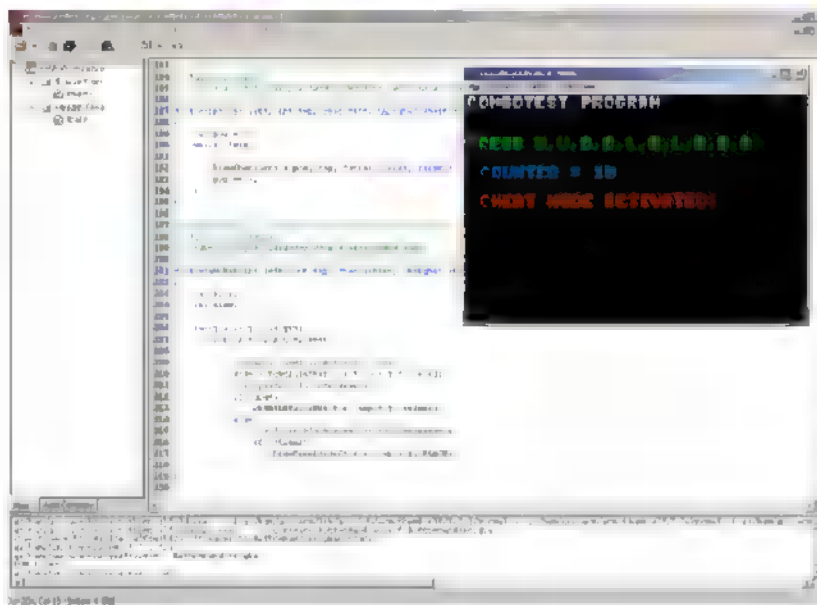
Figure 10.7

The ComboTest program shows how to use button combos.

Summary

This chapter has covered the absolutely critical subject of button input, and it wasn't too rough of a ride after all. Adding a button handler to a GBA program is probably even more important than the graphics or sound—in fact, all three are critical for ■ good game, so there really is no comparison for priority here. As you saw in this chapter, there's a lot you can do with a button handler, and I have only touched on the major points, such as detecting multiple button presses, displaying the button codes, and even handling combos.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter. .

Challenge 1: The ScanButtons3 program does a pretty good job of demonstrating button input, but it would be more interesting with more color. Modify the program so that a different color is used for the name of each button.

Challenge 2: The ButtonHandler program displays button press events on separate lines. Modify the program so it keeps a counter of each button pressed and displays the counter value with each button message on the screen.

Challenge 3: The ComboTest program currently only supports a single combo. Modify the program so that it can support many different combos and display the name of the combo when activated.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. How many buttons does the GBA have?

    A. 6

    B. 8

    C. 10

    D. 12

2. What is the memory addressed used to check the status of button input?

    A. 0x04000130

    B. 0x05020100

    C. 0x01000900

    D. 0x60000000

3. What is the type of processor used in the GBA?

    A. MIPS

    B. StrongARM

    C. SH3

    D. ARM7

4. True/False: Can the GBA can access 32 bits of memory at a time?

    A. True

    B. False

5. What method is used to check for button input on the GBA?

    A. Hardware interrupt

    B. Callback function

    C. Memory polling

D. Meditation

6. What is the largest value returned by any button press event?

    A. 128

    B. 256

    C. 512

    D. 1,024

7. How many total bits (out of 32) are used by the GBA to report button input status?

    A. 8

    B. 10

    C. 16

    D. 6

8. Which video mode is used by the sample programs in this chapter?

    A. Mode 0

    B. Mode 2

    C. Mode 5

    D. Mode 3

9. What bitwise operation is used to evaluate the status of a single button?

    A. AND

    B. OR

    C. NOT

    D. XOR

10. What bit value is set by the GBA to identify that a button press has occurred?

    A. 1

    B. 0

    C. -1

    D. 2

Chapter 11

ARM7 Assembly

Language Primer

This chapter is a brief overview of the ARM7 instruction set and assembly-language primer for the GBA. Assembly language is not difficult to understand, but mastering the subject requires time and patience. This chapter provides enough information for you to write ■ complete assembly-language program from scratch and also shows how to write assembly functions and call them from a main C program, which is most likely something that you will be doing sooner or later as you write complete GBA games.

For starters, you will learn how to compile a program from the command prompt using the HAM compiler chain manually, a necessary first step before assembling and linking the programs in the chapter. This is not a comprehensive chapter on ARM7 assembly language, by any means. In fact, it is rather sparse on the instruction set. The goal of this chapter is not to teach you how to write assembly language, but rather, how to use assembly to enhance a C program, with a few simple examples. Please refer to the reference books listed later in this chapter (as well as in Appendix B) that cover the ARM architecture.

Here is a rundown of the subjects in this chapter:

- Introduction to command-line compiling
- Basic ARM7 assembly language
- Calling assembly functions from C

Introduction to Command-Line Compiling

Before getting into assembly language, I would like to first explain how to compile ■ regular C program from the command line, because you will need this information in order to use the command-line tools. Visual HAM and the HAM SDK hide away the compiler chain quite well, which is primarily why I chose these tools for the book. Now that you have made it through the thick and thin of GBA coding and are ready for a little "pedal to the metal," I need to show you how to compile programs completely outside of Visual HAM--without even the benefit of a "make" file. A make file is a text file that describes the process of compiling, assembling, and linking source code files into a final binary .gba file, and this is what Visual HAM does when you press F5 or F7 to build the project, it invokes the make utility.

Have you noticed that new projects always come with a file called *"makefile"* with no extension? This is the default file that the make utility loads when you simply type "make" on the command line, with no options. It loads the default file and processes it. The make file has pathnames and specifies what to include in the compile options to take ■ simple C source file and produce a GBA ROM image out of it--which is no easy matter. It's just that HAM makes this very easy because it includes the complete compiler chain (all the utilities, assemblers, compilers, includes, and libs needed to build a GBA ROM).

Now what I'd like to do is take one of the sample programs from an earlier chapter and show you how to compile it manually. You'll then write a few short batch files that can be reused to compile other programs (including assembly-language files) into a .gba image.

Compiling from the Command Line

The TestBuild project on the CD-ROM was adapted from the simple DrawPixel program covered previously. This project is located in \Sources\Chapter11\TestBuild. Remember, you don't need to load this into Visual HAM, because this will be a command-line exercise! It's ■ very short program (without the comment lines), so I'll list the main.c file here. Use Notepad or ■ similar text editor to type in the program.

```
///////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 11: ARM7 Assembly Language Primer
// TestBuild Project
// main.c source code file
///////////////////////////////////////////


int main(void)
```

```
{
    //create pointer to video buffer
    unsigned short* videoBuffer = (unsigned short*)0x6000000;

    //enter video mode 3
    *(unsigned long*)0x4000000 = (0x3 | 0x400);

    //draw a white pixel centered on the screen
    videoBuffer[80 * 240 + 120] = 0xFFFF;

    while(1);
    return 0;
}
```

Creating a Compile Batch File

Compiling the program will require more work than typing in the program itself, unfortunately! But I'm going to make it so you only have to type in the compile options once in a batch file, which you can then reuse for the remaining programs in this chapter.

So, go ahead and open up Notepad again, type in the following, and then save the file as gcc.bat. I'll explain what it does after you have tried it out first. Although it takes up several lines in the listing, this is actually just one long line. If you have word wrap enabled in Notepad (via the Format menu), then just type in the compile command without any line breaks. Here is the entire command:

```
arm-thumb-elf-gcc.exe -I %HAMDIR%\gcc-arm\include -I %HAMDIR%\include
-I %HAMDIR%\gcc-arm\arm-thumb-elf\include -I %HAMDIR%\system -c -DHAM_HAM
-DHAM_MULTIBOOT -DHAM_ENABLE_MBV2LIB -O2 -DHAM_WITH_LIBHAM
-mthumb-interwork -mlong-calls -Wall -save-temps -fverbose-asm
%1.c -o%1.o
```

Don't forget the last short line--it's the most important one! Believe it or not, it takes that huge command just to compile a single C source file with GCC, due to all the include files and compiler directives that have been set up for GBA development.

Now let's try out the command. Bring up the Start menu in Windows, and select Programs, Accessories, Command Prompt. It should look like Figure 11.1.
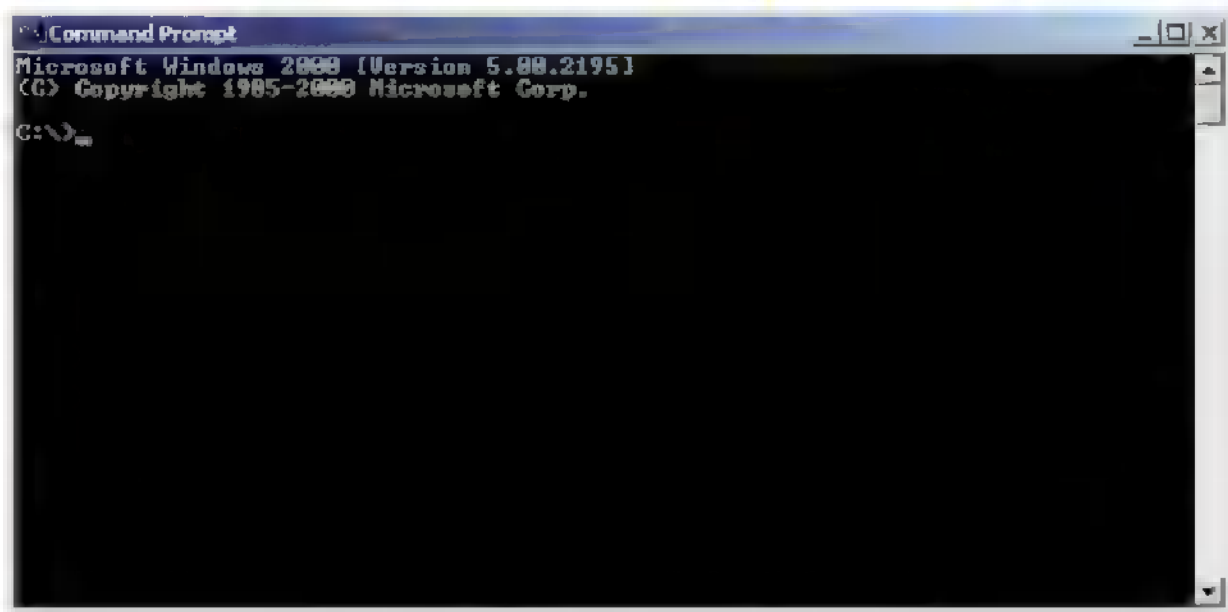
*Figure 11.1 - The Command Prompt as it
appears in a stock Windows 2000 system.*

Now, my GBA development tools and sources are all on another hard drive partition (G), so I'm going to switch over to that drive by typing "G:" and pressing Enter. I'll explain some of these steps because not everyone has a lot of experience with the command prompt (although I imagine some may recall the old MS-DOS days?). I've created ■ folder called TestBuild inside \GBA\Sources\Chapter11. If you have just copied the \Sources folder off the CD-ROM entirely, then ignore my specific folder names and follow your own configuration. I'm going to type
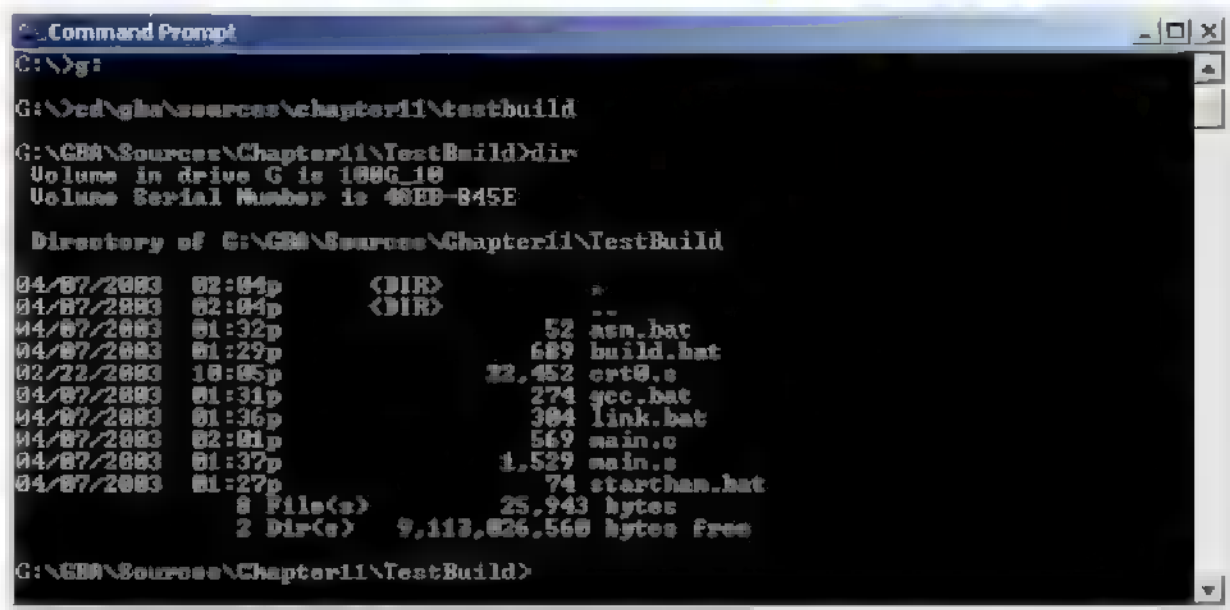
```
CD \GBA\Sources\Chapter11\TestBuild
```

to get into the correct folder for the TestBuild project. A list of files in this folder is shown in Figure 11.2, just to be sure we're in sync. If you have copied the sources off the CD-ROM to your hard drive, then you will want to replace the pathname here with the correct path to chapter 11 on your system.

Creating a Path to the Compiler Chain

Notice that there are some batch files in there other than the gcc.bat file you just created. One such file is startham.bat. This file is generated by HAM during installation and is located in \HAM. I have copied the file to the TestBuild folder so it's easier to use and edited the comments out of the file. Here is what it looks like:

```
set PATH=g:\ham\gcc-arm\bin;g:\ham\tools\win32;%PATH%

set HAMDIR=g:\ham
```

Figure 11.2 - The list of files inside the TestBuild project folder.

This batch file opens up a path to the HAM compiler chain and tools folders so you can invoke those tools from anywhere. The paths in this batch file should reflect your installation of HAM; if it differs, then be sure to correct the paths before running the batch file. Type **startham** now to set up the command-line paths.

Compiling The Program

You are now ready to compile the main.c file using gcc.bat, which you created ▪ moment ago. Here is how you compile the main.c file:

```
gcc main
```

Notice that I didn't include the .c extension. That's because the gcc.bat file adds the extension automatically. It needs the file name without the extension because it uses that name to generate the output file, which will be main.o. In the command-line realm, no news is good news. If no messages are printed on the command line, that means the GCC compiler compiled the program without any warnings or errors. Now, there will be a command displayed on the screen when you type "gcc main", because the batch file echoes the command on the screen. If you prefer to hide it, you can add **@echo off** to the top of the batch file, before the command, and that will hide the commands.

You can look in the TestBuild folder after running the command, and you should see a new file, main.o. This is the object file, containing machine instructions for the ARM7 chip in the GBA. These instructions are specific only to the program and do not know how to boot up or anything like that. There are a couple more steps involved before the main.o file can be converted to ▪ .gba file (a process called linking the object files). I will show you how to

Programming The Nintendo Game Boy Advance: The Unofficial Guide
Copyright (c)2003 by Jonathan S. Harbour -- http://www.jharbour.com

399

link the main.o file shortly. In the meantime, take a look at Figure 11.3, which shows an illustration of the compile process.

```
                  ┌─────────────────┐
                  │     main.c      │
                  │   source file   │
                  └─────────────────┘
                           │
                           ▼
                      (  GCC    )
                      ( compiler )
                       /        \
                      ▼          ▼
         ┌─────────────┐      ┌─────────────┐
         │   main.o    │      │   main.s    │
         │ object file │      │  assembly   │
         │             │      │    file     │
         │             │      │ (optional)  │
         └─────────────┘      └─────────────┘
```
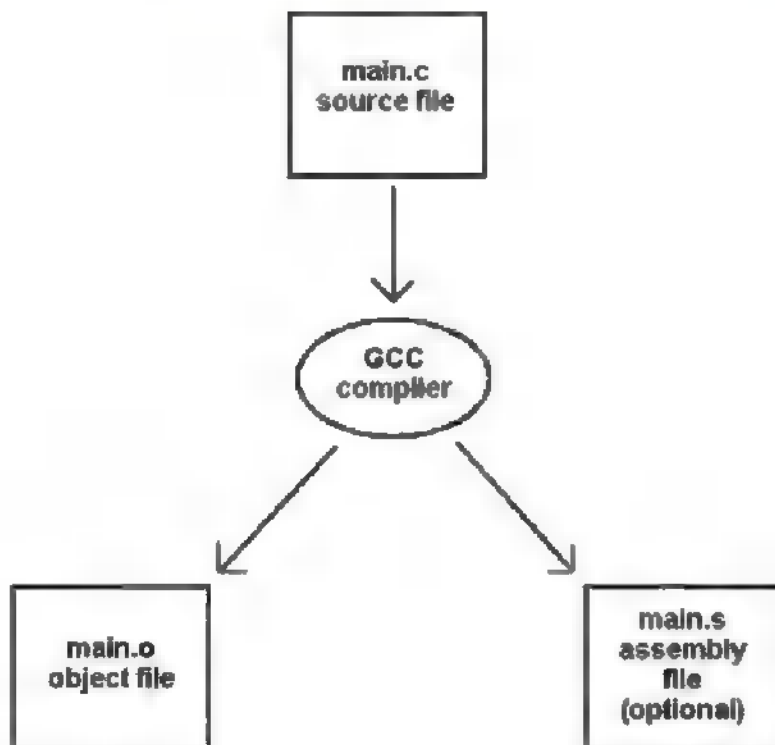
Figure 11.3

Compiling a source code file into an object file with the GCC compiler.

Assembling from the Command Line

In the GBA development realm, object files have an extension of .o. That is why the main.c file was compiled to main.o when you invoked **gcc main** a moment ago. The linker is another tool required to build a final .gba file, as it takes all the various object files and links them together into a single object file, which can then be turned into a .gba (using another command-line program that I will show you shortly). Before I give you the link command, though, one small step must first be done.

There is an assembler that comes with HAM and is located in the same place as the compiler. The exact file name of the assembler is **arm-thumb-elf-as.exe**, where the "as" means "assembler." Assembly language is the lowest programming language as far as being close to the hardware. Indeed, assembly is very difficult to master and is not for the faint of heart, because each assembly instruction translates directly to ∎ CPU instruction! So you are literally working directly with the innards of the CPU. Since this is a practical chapter rather than a theoretical one, I'm going to show you how to assemble a file without really explaining what is in that file. The file is called crt0.s, and it contains all the bootstrap and execution code needed to build a GBA ROM image. This crt0.s file must be assembled and

linked together with your main.o object file in order to run it on a GBA (or inside an emulator).

The crt0.s file is a bootstrap source code file that provides all the services needed by ■ GBA program, such as GBA initialization. The crt0.s assembly file is what you might call the boot sector program. On a PC, there is a boot sector on each hard drive, and the very first sector on the hard drive includes operating system bootup instructions. The BIOS (basic input-output system) on the PC, after checking over the hardware on the PC, will invoke this small program on whatever drive is available. Usually, the first drive is a floppy drive, which will try to boot if you insert a disk when powering up the PC. On most PCs, the CD-ROM is also bootable, although it is the hard drive that boots most of the time. The operating system installs a boot sector program that is run by the BIOS, and this boot sector program will run an operating system loader. For instance, the operating system loader for Windows is command.com, and has been that same filename since the very first version of MS-DOS. The crt0.s program is similar to the bootstrap program on a PC, only it is designed to boot up the GBA. Remember, the ROM image in your .gba program is all the GBA has to go by, as there is no operating system on the GBA, so the bootstrap must be included in the ROM!

I have copied the crt0.s assembly file out of the HAM folder and placed it inside the TestBuild project folder, because I want to eliminate long pathnames as much as possible. To assemble the file, you will want to type in the following command:

```
arm-thumb-elf-as.exe -mthumb-interwork crt0.s -ocrt0.o
```

The reason why this file name is so long is because it's descriptive. The ARM7 processor has two different modes built in, and you may switch between them at any time. The modes are ARM and THUMB. ARM is the full 32-bit instruction set, while THUMB is a 16-bit hardware-emulated instruction set. While the programs in this chapter do no use any thumb code, I want to include support for this mode because you may want to reuse the commands stored in the gcc.bat batch file (and the other batch files in this chapter).

Creating an Assemble Batch File

As usual, there will be no message if the file was assembled correctly without error, and there will be a new file called crt0.o in the TestBuild folder, so go ahead and take ■ look. You now have the boot object file and your main object file and are ready to link. But first I want to turn that assembly command into a more convenient batch file that can be reused (as it will be later).

Create a new text file called asm.bat and type the following line into this file:

```
arm-thumb-elf-as.exe -mthumb-interwork %1.s -o%1.o
```

This is the same command, essentially, but the file name has been replaced with a parameter, %1, that will fill in the file name passed to it. Now, using this batch file, you can

assemble any .s file, (for instance: **asm crt0**), noting again the lack of an extension, as it is automatically filled in. Figure 11.4 illustrates how the assembler works.
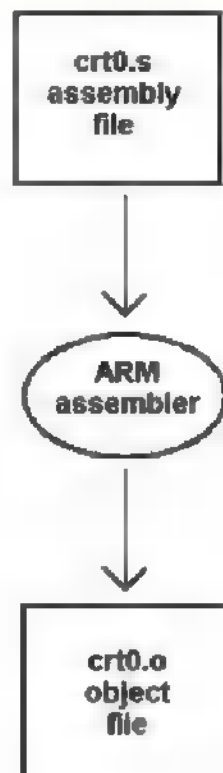

Figure 11.4

Assembling an assembly language file into an object file with the ARM assembler.

Linking from the Command Line

The two object files have been created--one from the main.c source file, the other from the crt0.s assembly-language file--and are ready to be linked together. The linking process is more involved, because a lot of libraries must be included in order to satisfy all the function calls within crt0.o and main.o. I'm not going to spend much time explaining all the libraries because they are just part of the GCC compiler chain for the GBA.

Creating a Linker Batch File

Okay, let's get started, this time with the batch file right away. Because I only want to go over the command itself once, we might as well just stick it into a batch file. Create a new file and call it link.bat, typing the following command into the file. Again, if you're using Notepad, simply type away and don't fill in any new lines, as this should be a single long command. After typing in the link command, there is one more command to be added to this batch file, a call to objcopy that actually takes the linked .elf file and converts it to a .gba file. So these two commands are both inside the link.bat file.

```
@echo off
arm-thumb-elf-ld.exe -L %HAMDIR%\gcc-arm\lib\gcc-lib\arm-thumb-elf\3.2.2\normal
-L %HAMDIR%\gcc-arm\arm-thumb-elf\lib\normal -L %HAMDIR%\gcc-arm\lib
--script %HAMDIR%\system\lnkscript-afm -o%1.elf %1.o crt0.o -lafm -lham -lm
-lstdc++ -lsupc++ -lc -lgcc


arm-thumb-elf-objcopy.exe -v -O binary %1.elf %1.gba
```

Make sure these two commands are both inside the link.bat file, and typed without using the Enter key, and then you can link the program with the following command:

```
link main
```

If all goes well, you should see the commands echoed to the screen, but otherwise no error messages. Which means. . .you now have compiled your first GBA program the hard way! The program should look like Figure 11.5. If the link process worked without error, you should see a main.gba file, which you can load into the emulator.
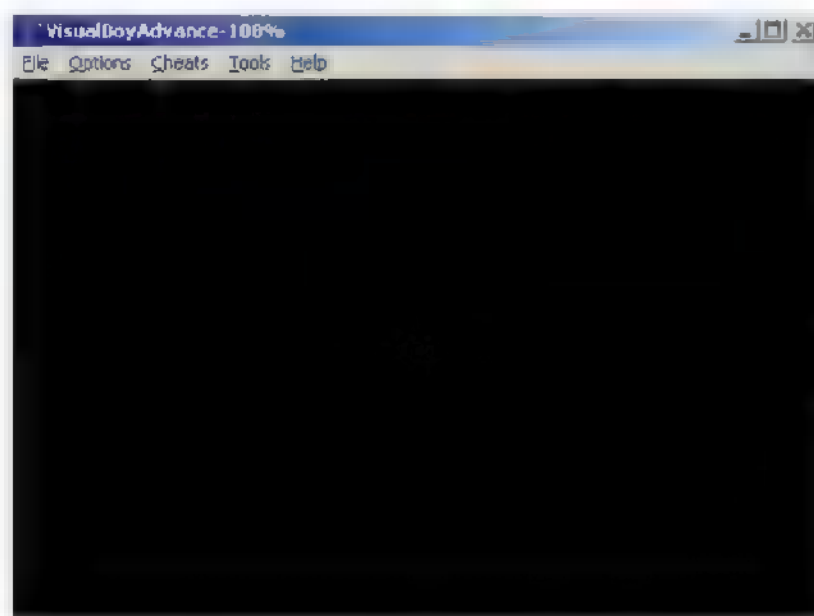
Figure 11.5
The TestBuild program draws a pixel as expected.

If instead you get an error message related to an unrecognized command, the problem is most likely due to a linebreak in the batch file. Make sure that each of the two commands are free of linebreaks. You may also double check to ensure that you ran startham.bat first, in order to set up the environment variables for the command-line tools.

Keep the batch files you have created--asm.bat, gcc.bat, and link.bat--handy, as you'll need them for the next program. It's incredibly beneficial to understand how the compiler, assembler, and linker works, so this experience will be valuable to you in your GBA coding

efforts. Don't rely solely on Visual HAM and the HAM SDK to do all the work for you. After all, HAM was originally designed to be run from the command line in order to build GBA programs. Visual HAM came later. Figure 11.6 illustrates the process of linking object files into an executable file.
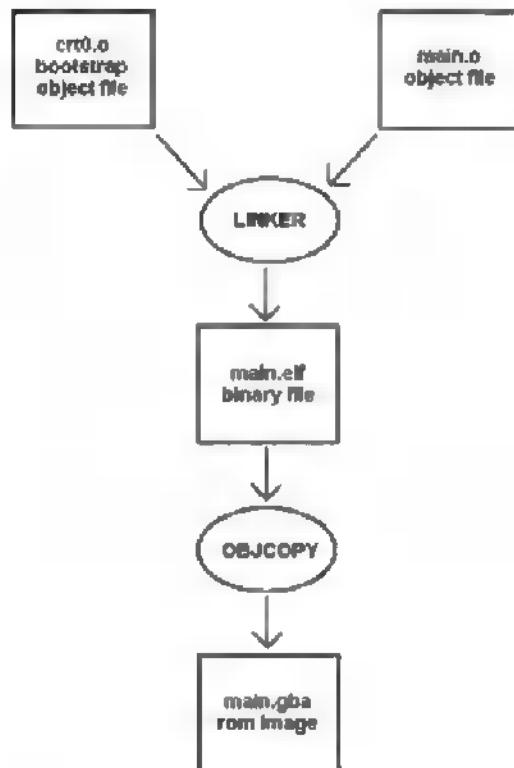

Figure 11.6

Converting object files into an executable rom image file.

(Very) Basic ARM7 Assembly Language

In this section I'm going to walk you through two very simple assembly-language programs help you gain a little appreciation for the C language, for one thing, but also to familiarize you with what an assembly file looks like and how you can start to enhance your GBA programs with very low-level code. Unfortunately, ARM7 assembly language is a complicated subject, and I am only showing you *what* it is in this chapter, rather than going into any detail about *how* to use it to the fullest extent. For details on the ARM7TDMI CPU and its instruction set, you may refer to an online reference (go to www.google.com and search for "arm7tdmi"), or you may order a reference book (see Appendix B).

There is an excellent tutorial on ARM7 assembly language on the Web at http://k2pts.home.attbi.com/gbaguy, and another great online reference guide at http://re-eject.gbadev.org/. Just keep in mind that nothing on the Web is permanent, and these URLs are subject to change.

Assembly instructions are very similar to the CPU instructions, which is why you'll see mov, add, str, b, and other obscure statements, often combined with one or more registers. That's also why assembly is such a difficult language to master. The ARM7 chip has many general-purpose registers, such as r1, r2, r3, r4, r5, and so on. While it's true that once you have learned assembly for one processor, you have a good chance of quickly learning assembly language for other processors, the architectures can vary widely. For instance, the ARM7 is a reduced instruction set computer (RISC) chip, while common PC processors from Intel, AMD, and others are complex instruction set computer (CISC) chips.

The difference in these architectures is significant but may be broken down into two main areas: registers and instructions. A RISC chip has many registers but few instructions. A CISC chip has only a few registers but many instructions. The supposed extra performance in a RISC chip is due in part to the long pipelines in many processors today. The Intel Pentium 4 processor has a 20-stage pipeline. If this were a RISC chip, it would be much faster, because branch prediction would be more accurate (due to the many available registers and fewer instructions). The ARM7 has a 3-stage pipeline, which is very good for a low-voltage and small footprint processor (that runs on two AA batteries!). These are all subjects relegated to a hardware discussion, and I mention them simply to make a point, so don't worry if you are unfamiliar with hardware terminology, it's not important for writing GBA code.

A Simple Assembly Program

The really practical aspect of this chapter is that I want to show you not a list of the ARM7 instruction set, but rather only a few useful instructions that can be used inside a given function to enhance a C program. For starters, I'll show you how to write your first assembly-language program that will run on the GBA. This program is called FirstAsm and is shown running in Figure 11.7. Not very impressive, but it's accessing the video buffer and drawing a pixel, all in 100 percent assembly language. Are you intrigued? It's *a lot* of fun writing your first assembly program! Here we go.
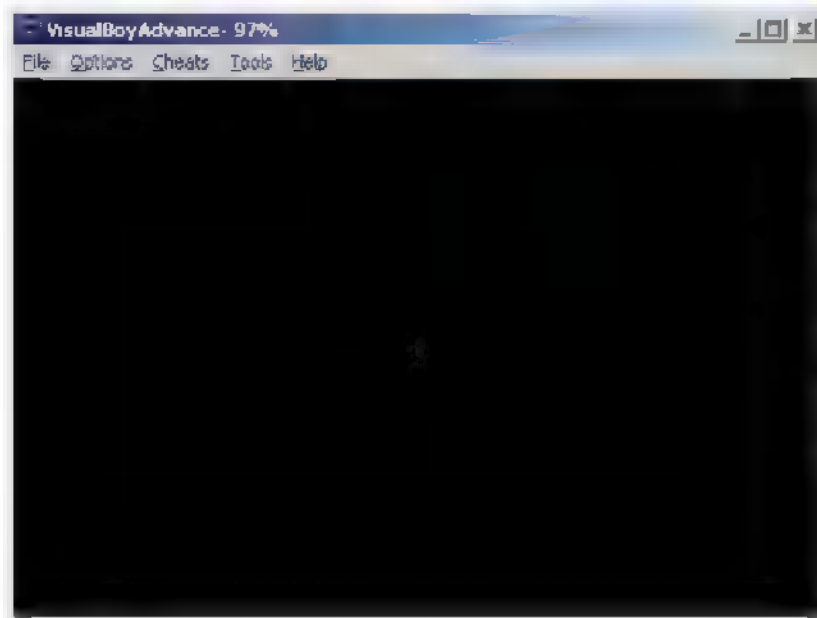
Figure 11.7

The FirstAsm program draws a single pixel in the center of the screen.

The FirstAsm Program

The FirstAsm program is a simple assembly listing that sets the video mode to mode 3 with background 2 enabled and then draws a pixel at the center of the screen. I have included comments with each line. As you can see from the listing that follows, a comment in ARM7 assembly starts with the @ character. Anything that falls on the same line after that character is ignored by the assembler. Now, you are going to want to type this code into a file called pixel.s.

```
@//////////////////////////////////////////////
@ Programming The Game Boy Advance
@ Chapter 11: ARM7 Assembly Language Primer
@ FirstAsm Program
@ pixel.s assembly file
@//////////////////////////////////////////////

    .text
    .align2
    .globalmain

@main entry point of program
main:
```

```
@set video mode 3 using background 2

    mov    r2, #1024           @BG2_ENABLE

    add    r2, r2, #3          @mode 3

    mov    r3, #67108864       @REG_DISPCNT

    str    r2, [r3, #0]        @set value


@draw a white pixel at 80x120
@remember, mode 3 is 2 bytes/pixel

    mov    r1, #38400@80*240*2

    add    r1, r1, #240@X=120

    add    r3, r3, #33554432@videoMemory

    mvn    r2, #0  @draw pixel

    strh   r2, [r3, r1]


@endless loop
.forever:

    b    .forever


@define object size of main
.end:

    .sizemain,.end-main
```

All done? Great! Now let's assemble and run this baby. If you created the batch files that I covered earlier, then you should have an asm.bat file available. You may want to copy the asm.bat file and pixel.s files into a new folder that is just for this project. Or feel free to just leave this file with the other files from this chapter, in a single folder. That way you can just use the batch files, and crt0 file will be readily available. On the CD-ROM, these files are all in separate project folders; if you want to just copy them off the CD, feel free to do so. Now, from the command prompt, assuming you are in the correct folder, type this:

```
asm pixel
```

to assemble the source code file for the program. If there were no errors, then you can link the file and run it. I'm assuming you already assembled the crt0.s file earlier in this chapter. If not, refer to the start of the chapter for a tutorial on assembling this file and why it is needed. Now let's link:

```
link pixel
```

That's all there is to it! You should now see a pixel.gba file in the folder. Go ahead and run it in VisualBoyAdvance, just as you have done for all the previous projects in the book. Congratulations! You have just written your first ARM7 assembly-language program.

Calling Assembly Functions from C

The real goal of this chapter is to show you how you can use assembly functions to optimize your GBA games, presumably written in C. To do that, you will need to write the assembly code in a separate .s file and then define an external function prototype in the C source code file. For this example program, which I have called ExternAsm, I will show you how to write a simple DrawPixel function in assembly and then use it from a C program. To keep things simpler on the asm side, I have passed the video buffer to the asm function as a parameter. The function prototype looks like this:

```
extern void DrawPixel32 (u32 x, u32 y, u32 color, u32 videobuffer);
```

Can you imagine the power this gives you? If you learn a little ARM7 assembly language, you can optimize any part of your code that is written in C, making use of the astounding speed of assembly language. Of course, you could even write the entire game in assembly, although I wouldn't recommend it. I knew someone who wrote a Sega Genesis game entirely in 68000 assembly, and it was not a fun experience for him. For one thing, simple things like loading data from a file is an enormous chore in assembly, and the resulting benefits for system-level functionality like that are more sensibly utilized in C, reserving assembly for optimization. That is the method of most game programmers--and it is what I recommend.

Making the Function Call

The actual function prototype is all you need to declare an external function in C. Isn't that easy? Imagine just replacing slow C functions in your game with assembly by simply adding an external function prototype! Here's the source code for the extern.c file, the main C source code file for the ExternAsm program. I will go over the actual DrawPixel32 assembly function next.

```
/////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 11: ARM7 Assembly Language Primer
// ExternAsm Project
// extern.c source code file
/////////////////////////////////////////////////

typedef unsigned long u32;
```

```
//declare prototype for the external assembly function
extern void DrawPixel32 (u32 x, u32 y, u32 color, u32 videobuffer);


//video mode register
#define REG_DISPCNT *(unsigned long*)0x4000000


int main(void)
{
    u32 x, y;

    //set video mode 3
    REG_DISPCNT = (3 | 0x400);


    //fill screen with a pattern
    for (y = 0; y < 159; y++)
        for (x = 0; x < 239; x++)
            DrawPixel32(x, y, x*y%31, 0x6000000);


    while(1);
    return 0;
}
```

The DrawPixel32 Assembly Code

Now for the DrawPixel32 source code. This function was written in ARM7 assembly language, and as you can see, it is very short. Since all four parameters are unsigned int (u32) data types, it was a simple matter to use the registers directly without any intervention code (for instance, to move an 8- or 16-bit number into a 32-bit register, and vice versa). Create a new text file called drawpixel.s and type this code listing into the file, then save it.

```
@ Draw pixel in GBA graphics mode 3
@ DrawPixel32(u32 x, u32 y, u32 color, u32 videobuffer);
@ r0 = x
@ r1 = y
@ r2 = color
@ r3 = videobuffer
```

```
        .ARM

        .ALIGN

        .GLOBL   DrawPixel32

DrawPixel32:

        stmfd    sp!,{r4-r5}      @ Save register r4 and r5 on stack

        mov      r4,#480          @ r4 = 480

        mul      r5,r4,r1         @ r5 = r4 * y

        add      r5,r5,r0,lsl #1  @ r5 = r5 + (x << 1)

        add      r4,r5,r3         @ r4 = r5 + videobuffer

        strh     r2,[r4]          @ *(unsigned short *)r4 = color

        ldmfd    sp!,{r4-r5}      @ Restore registers r4 and r5

        bx       lr
```

Compiling the ExternAsm Program

To compile the ExternAsm program, you will need to compile the extern.c, and assemble the drawpixel.s file, and then link them both. The link.bat file can't accommodate two object files, so I have modified it to accept two object files (by simply adding %2.o to the command;, see the link2.bat file in the ExternAsm folder). If you ever write ■ program with numerous object files (*.o), then you may need to modify the link.bat file again to accommodate as many .o files as you need. There are ways to add conditional code to ■ batch file to accommodate as many parameters as are passed to it, but the batch code is somewhat involved, and I don't want to get into it at this point, when this batch file in particular needs only two parameters.

To compile the extern.c file:

```
gcc extern
```

To assemble the drawpixel.s file:

```
asm drawpixel
```

And then, to link them together into a runnable .gba file:

```
link2 extern drawpixel
```

If the compiler, assembler, and linker all returned with no errors, then you have successfully built your first assembly-enhanced program! Quick, run the program in VisualBoyAdvance to see what it looks like. The program's output is shown in Figure 11.8.

Now that's more like it! This program does a lot more than the simple pixel plotter from the last two programs! This program actually fills in the screen with an attractive pattern that is generated entirely using the x and y variables, resulting in what I like to call the red rose effect. Kind of strange, huh? Well, it just goes to show that weird things can happen when you're having fun.
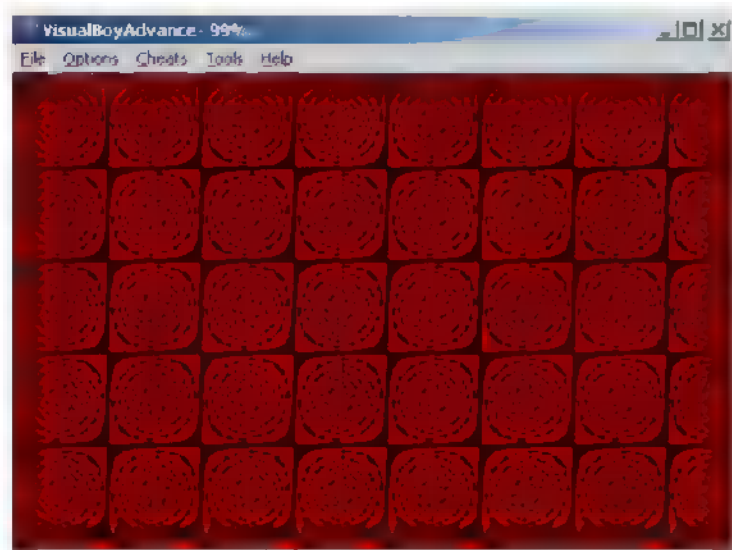
Figure 11.8
The ExternAsm program demonstrates how to call an external assembly function from a C program.

Summary

Well, this has been a fun chapter, but I must admit that there was a lot more that I wanted to cover. The one thing you *don't* want to hear is "Yeah, yeah, it's *beyond the scope of the chapter.*" But honestly, that is the truth. I encourage you to learn more about the ARM7 chip that powers the GBA and to learn the instruction set.

To master ARM7 assembly language is to master the GBA, no doubt about it. You will also likely find yourself in gainful employment with a GBA developer, because skilled C programmers who are knowledgeable of the low-level assembly language on a given platform are in high demand.

For starters, I recommend you write a complete game, and then look for ways to optimize it. Look first to your C code, and make sure it is as tight as possible. Then look for bottlenecks that can be improved with assembly. You would be surprised by how even the simplest function implemented in assembly can have a drastic impact on the performance of a game. Good luck!

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: The FirstAsm program draws a pixel in the very center of the screen. See if you can modify the assembly code to have it draw the pixel somewhere else on the screen.

Challenge 2: The ExternAsm program demonstrates how to call an external assembly function. See if you can modify the DrawPixel32 function so that it moves through video memory (which is 38,400 bytes long) and fills the entire screen with a pixel. You may then rename the function to FillScreen32.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in Appendix D.

1. What extension does an assembly-language file have?

    A. .S

    B. .ASM

    C. .AL

    D. .C

2. What is the 16-bit instruction set on the ARM7 chip called?

    A. ARM

    B. THUMB

    C. HAND

    D. FINGER

3. What is the 32-bit instruction set on the ARM7 chip called?

    A. ARM

    B. THUMB

    C. HAND

    D. FINGER

4. What is the full name of the GCC compiler?

   A. thumb-arm-elf-gcc.exe

   B. if-then-else-gcc.exe

   C. arm-thumb-elf-gcc.exe

   D. open-source-gcc.exe

5. What is the full name of the ARM7 assembler?

   A. hand-finger-thumb-as.exe

   B. hobbit-wizard-orc-as.exe

   C. leg-foot-toe-as.exe

   D. arm-thumb-elf-as.exe

6. What is the extension of the file generated by the linker?

   A. .HBT

   B. .ELF

   C. .ORC

   D. .HMN

7. What does the arm-thumb-elf-objcopy.exe program do?

   A. It links the bitmap and sound files into the main program object file.

   B. It copies an individual object out of a .ELF file into a C array.

   C. It links the object files together into a single .ELF file.

   D. It converts the .ELF object file into a .GBA ROM image file.

8. What does RISC stand for?

   A. Reduced Instruction Set Computer

   B. Really Ignorant Stupid Computer

   C. Radically Integrated Super Computer

   D. Recognition in Some Company

9. How are variables passed from a C calling function to an assembly function?

A. Interrupts

B. Registers

C. Stack

D. DMA

10. True or False: The ARM7 processor is a CISC chip.

A. True

B. False

Epilogue

This book has been, without a doubt, the most enjoyable book I have written so far. Getting down to the bare metal of a console has been an absolute blast, and I am grateful to have been blessed with the opportunity to write this book. I hope you have enjoyed it too! While this has not been a comprehensive reference of the Game Boy Advance, by any means, I believe this book succeeds in the goal I set out for it at the start--to teach anyone of any experience level how to write their own games for the Game Boy Advance. What an experience it has been!

Although I do not know you personally, I have gotten to know many readers and fans of my other books through online forums, so there is a certain feeling of coming full circle at this point. I hope you have found this book not just helpful, but invaluable as a reference, and enjoyable to read. I have strived to cover all the bases of this subject within the context of the goals for this book, and hope you have enjoyed it. There is much more to be learned, and the Game Boy Advance is capable of much, much, much more than what I have presented here! That small ARM7 processor is powerful and can handle fully textured 3D rendering, although that requires a software implementation. I encourage you to seek out the many excellent sample programs and demos written by fans online.

Although every effort was made to ensure that the content and source code presented in this book is error-free, it is possible that errors in print or bugs in the sample programs might have missed scrutiny. If you have any problems with the source code, sample programs, or general theory in this book, please let me know! You can contact me at

support@jharbour.com

and I'll do my best to help you work though any problems. I also welcome constructive criticism and comments that you might have regarding the book in general, or a specific aspect of the book. I get several hundred e-mails a month from readers and respond to every one!

Finally, whether you are an absolute beginner or a seasoned professional, I welcome you to join the discussion list on YahooGroups, where you will have an opportunity to share your games, ideas, and questions with other Game Boy Advance fans! Membership is free and open to the public. Just send an e-mail to the list server at

hamdev@yahoogroups.com

or visit the web site at http://www.yahoogroups.com, and search for the list by name. The list is maintained by Emanuel Schleussinger, the person who created HAM.

Of course, I also recommend you visit my Web site at

http://www.jharbour.com

to keep up to date on the Game Boy Advance development community and any changes or bug reports for the code presented in this book. As always, I look forward to hearing from you!

Part IV

The Mother Of All Appendices

Welcome to Part IV of *Programming The Game Boy Advance*. Part IV includes five appendices that provide reference information for your use, including an ASCII chart, a list of helpful books and Web sites, and an overview of the included CD-ROM, answers to the chapter quizzes, and even a Game Boy Advance hardware reference.

- Appendix A: ASCII Chart

- Appendix B: Recommended Books and Web Sites

- Appendix C: Game Boy Advance Hardware Reference

- Appendix D: Answers To The Chapter Quizzes

- Appendix E: Using The CD-ROM

Appendix A

ASCII Chart

This is a standard ASCII chart of character codes ▓ to 255. To use an ASCII code, simply hold down the Alt key and type the appropriate value to insert the character.

null	000	←	027	6	054	Q	081
☺	001	└	028	7	055	R	082
●	002	↔	029	8	056	S	083
♥	003	▲	030	9	057	T	084
♦	004	▼	031	:	058	U	085
♣	005	space	032	;	059	V	086
♠	006	!	033	<	060	W	087
•	007	"	034	=	061	X	088
▫	008	#	035	>	062	Y	089
○	009	$	036	?	063	Z	090
■	010	%	037	@	064	[	091
♂	011	&	038	A	065	\	092
♀	012	'	039	B	066	]	093
♪	013	(	040	C	067	^	094
♫	014	)	041	D	068	_	095
☼	015	*	042	E	069	`	096
►	016	+	043	F	070	a	097
◄	017	,	044	G	071	b	098
↕	018	-	045	H	072	c	099
‼	019	.	046	I	073	d	100
¶	020	/	047	J	074	e	101
§	021	0	048	K	075	f	102
▬	022	1	049	L	076	g	103
↨	023	2	050	M	077	h	104
↑	024	3	051	N	078	i	105
↓	025	4	052	O	079	j	106
→	026	5	053	P	080	k	107

l	108	ë	137	ª	166	├	195
m	109	è	138	º	167	─	196
n	110	ï	139	¿	168	┼	197
o	111	î	140	⌐	169	╞	198
p	112	ì	141	¬	170	╟	199
q	113	Ä	142	½	171	╚	200
r	114	Å	143	¼	172	╔	201
s	115	É	144	¡	173	╩	202
t	116	æ	145	«	174	╦	203
u	117	Æ	146	»	175	╠	204
v	118	ô	147	░	176	═	205
w	119	ö	148	▒	177	╬	206
x	120	ò	149	█	178	╧	207
y	121	û	150	│	179	╨	208
z	122	ù	151	┤	180	╤	209
{	123	ÿ	152	╡	181	╥	210
		124	Ö	153	╢	182	╙	211
}	125	Ü	154	╖	183	╘	212
~	126	¢	155	╕	184	╒	213
⌂	127	£	156	╣	185	╓	214
Ç	128	¥	157	║	186	╫	215
ü	129	Pts	158	╗	187	╪	216
é	130	ƒ	159	╝	188	┘	217
â	131	á	160	╜	189	┌	218
ä	132	í	161	╛	190	█	219
à	133	ó	162	┐	191	▄	220
å	134	ú	163	└	192	▌	221
ç	135	ñ	164	┴	193	▐	222
ê	136	Ñ	165	┬	194	▀	223

α	224	²	253
ß	225	■	254
Γ	226	blank	255
π	227		
Σ	228		
σ	229		
µ	230		
τ	231		
Φ	232		
Θ	233		
Ω	234		
δ	235		
∞	236		
φ	237		
ε	238		
∩	239		
≡	240		
±	241		
≥	242		
≤	243		
⌠	244		
⌡	245		
÷	246		
≈	247		
°	248		
·	249		
·	250		
√	251		
ⁿ	252		

Appendix B

Recommended Books And Web Sites

The following books and Web sites are invaluable learning and reference tools that cover programming, video game history, and more.

Recommended Books

Here is a list of good programming books, including some of my favorites and some that I have written myself. You may find a few to be helpful when learning Game Boy programming. Along with beginner books, this list presents advanced books, as well as some references for the ARM chip.

ARM Architecture Reference Manual (2nd Edition) (2000)

Dave Jagger. Addison-Wesley Publishing Company. ISBN 0201737191.

This book is an excellent resource for the ARM processor in all of its variations.

ARM System-On-Chip Architecture (2nd Edition) (2000)

Stephen B. Furber. Addison-Wesley Publishing Company. ISBN 0201675196.

This book describes how to design a CPU system-on-chip around a microprocessor core, using the ARM architecture as a case study.

Beginner's Guide to DarkBASIC Game Programming (2002)

Jonathan S. Harbour and Joshua R. Smith. Premier Press. ISBN 1-59200-009-6.

This book provides a good introduction to programming Direct3D, the 3D graphics component of DirectX, using the C language.

C Programming for the Absolute Beginner (2002)

Michael A. Vine. Premier Press. ISBN 1-931841-52-7.

This book teaches C programming using the free GCC compiler as its development platform, which is the same compiler used to write Game Boy programs! As such, I highly recommend this starter book if you are just learning the C language. It sticks to just the basics. You will learn the fundamentals of the C language without any distracting material or commentary, just the fundamentals of what you need to be a successful C programmer.

C++ Programming for the Absolute Beginner (2001)

Dirk Henkemans and Mark Lee. Premier Press. ISBN 1-931841-43-8.

If you are new to programming with C++ and you are looking for a solid introduction, this is the book for you. This book will teach you the skills you need for practical C++ programming applications and how you can put these skills to use in real-world scenarios.

Game Design: The Art & Business of Creating Games (2001)

Bob Bates. Prima Tech. ISBN 0-7615-3165-3.

This very readable and informative book is a great resource for learning how to design games[--]the high-level process of planning the game prior to starting work on the source code or artwork.

Game Programming All in One (2002)

Bruno Miguel Teixeira de Sousa. Premier Press. ISBN 1-931841-23-3.

This book presents everything you need to get started as a game developer using the C language. Divided into increasingly advanced sections, it covers the most important elements of game development. Beginners start with the basics of C programming early in the book. Later chapters move on to Windows programming and the main components of DirectX.

High Score! The Illustrated History of Electronic Games (2002)

Rusel DeMaria and Johnny L. Wilson. McGraw-Hill/Osborne. ISBN 0-07-222428-2.

This gem of a book covers the entire video game industry, including arcade machines, consoles, and computer games. It is jam-packed with wonderful interviews with famous game developers and is chock-full of color photographs, including detailed information about Nintendo and the development of the Game Boy Advance.

Microsoft C# Programming for the Absolute Beginner (2002)

Andy Harris. Premier Press. ISBN 1-931841-16-0.

Using game creation as a teaching tool, this book teaches not only C# but also the fundamental programming concepts you need to grasp to learn any computer language. You will be able to take the skills you learn from this book and apply them to your own situations. *Microsoft C# Programming for the Absolute Beginner* is a unique book aimed at the novice programmer. Developed by computer science instructors, this series is the ideal tool for anyone with little to no programming experience.

Microsoft Visual Basic .NET Programming for the Absolute Beginner (2002)

Jonathan S. Harbour. Premier Press. ISBN 1-59200-002-9.

Whether you are new to programming with Visual Basic .NET or you are upgrading from Visual Basic 6.0 and looking for a solid introduction, this is the book for you. It teaches the basics of Visual Basic .NET by working through simple games that you will learn to create. You will acquire the skills you need for more practical Visual Basic .NET programming applications and learn how to put these skills to use in real-world scenarios.

Pocket PC Game Programming: Using the Windows CE Game API (2001)

Jonathan S. Harbour. Premier Press. ISBN 0-7615-3057-6.

This book will teach you how to program a Pocket PC handheld computer using Visual Basic or Visual C++. It includes coverage of graphics, sound, stylus and button input, and even multiplayer capability. Numerous sample programs and games demonstrate the key topics needed to write complete Pocket PC games.

Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games (2001)

Neal and Jana Hallford. Prima Tech. ISBN 0-7615-3299-4.

This book is a fascinating overview of what it takes to develop a commercial-quality role-playing game, from design to programming to marketing. This is a helpful book if you would like to write a game like *Zelda*.

Visual Basic Game Programming with DirectX (2002)

Jonathan S. Harbour. Premier Press. ISBN 1-931841-25-X.

This book is a comprehensive programmer's tutorial and a reference for everything related to programming games with Visual Basic. After a complete explanation of the Windows API graphics device interface meant to supercharge 2D sprite programming for normal applications, the book delves into DirectX 7.0 and 8.1 and covers every component of DirectX in detail, including Direct3D. Four complete games are included, demonstrating the code developed in the book.

Recommended Web Sites

Following is a list of Web sites that you will find useful when you are learning to program the Game Boy Advance and also as you start writing your own games. For the latest updates to the Web site list,

including links to new Web sites dedicated to the Game Boy Advance, please visit my Web site at http://www.jharbour.com and click the **Books** link to find the official site for this book.

Code Waves

http://www.codewaves.com

The home site of the GBA sound library and other tools.

CowBite Virtual Hardware Specifications

http://www.cs.rit.edu/%7Etjh8300/CowBite/CowBiteSpec.htm

A detailed and invaluable hardware reference for the GBA.

Game Boy Advance Dev'rs

http://www.devrs.com/gba

A useful programming site that focuses on GBA development, including links to GBA tools.

HAM and Hamlib

http://www.ngine.de

The home site for HAM and Hamlib, the GBA development distribution kit included and used in this book.

Microsoft DirectX

http://www.microsoft.com/directx

Microsoft's main DirectX site, where you can download the latest version of DirectX. You will need DirectX in order to run the VisualBoyAdvance emulator.

Visual HAM Home Page

http://visualham.console-dev.de

The home site for the Visual HAM integrated development environment used to write Game Boy programs in this book.

GameDev.net

http://www.gamedev.net

A well-respected online resource for all things related to game development.

Jonathan S. Harbour: Author's Home Page

http://www.jharbour.com

Jonathan's home page, with downloads, links, and resources for this book.

Nintendo Home

http://www.nintendo.com

The primary portal for all Nintendo products.

Nintendo Company History

http://www.nintendo.com/corp/history.html

The source of all historical references used in this book.

Appendix C

Game Boy Advance Hardware Reference

Throughout the book there have been source code listings that made use of standard defines, memory address values, and constants needed to write GBA programs. Here is a complete reference of all those lists in one convenient location. There were some cases where I defined values with slightly different names in the text of the book in order to clarify or explain the purpose of a register or value more easily. The important thing is to know when and how to use these values, rather than being overly specific on naming conventions.

Multiboot

```
#define MULTIBOOT int __gba_multiboot;
```

Bit Values

```
#define BIT00   1
#define BIT01   2
#define BIT02   4
#define BIT03   8
#define BIT04   16
#define BIT05   32
#define BIT06   64
#define BIT07   128
#define BIT08   256
#define BIT09   512
#define BIT10   1024
#define BIT11   2048
#define BIT12   4096
#define BIT13   8192
#define BIT14   16384
#define BIT15   32768
```

Typedefs

```
typedef unsigned char          u8;
typedef unsigned short         u16;
typedef unsigned long          u32;
typedef signed char            s8;
typedef signed short           s16;
typedef signed long            s32;
```

```
typedef unsigned char          byte;
typedef unsigned short         hword;
typedef unsigned long          word;
typedef volatile unsigned char     vu8;
typedef volatile unsigned short    vu16;
typedef volatile unsigned long     vu32;
typedef volatile signed char       vs8;
typedef volatile signed short      vs16;
typedef volatile signed long       vs32;
```

Buttons

```
volatile u32* BUTTONS = (volatile u32*)0x04000130;
#define BUTTON_A        1
#define BUTTON_B        2
#define BUTTON_SELECT   4
#define BUTTON_START    8
#define BUTTON_RIGHT    16
#define BUTTON_LEFT     32
#define BUTTON_UP       64
#define BUTTON_DOWN     128
#define BUTTON_R        256
#define BUTTON_L        512
```

Sprites

```
#define OAMmem          (u32*)0x7000000
#define OAMdata         (u16*)0x6100000
#define OBJPaletteMem   (u16*)0x5000200

//Attribute0 values
#define ROTATION_FLAG      0x100
#define SIZE_DOUBLE        0x200
#define MODE_NORMAL        0x0
#define MODE_TRANSPARENT   0x400
#define MODE_WINDOWED      0x800
```

```
#define MOSAIC              0x1000
#define COLOR_16            0x0000
#define COLOR_256           0x2000
#define SQUARE              0x0
#define WIDE                0x4000
#define TALL                0x8000


//Attribute1 values
#define ROTDATA(n)          ((n) << 9)
#define HORIZONTAL_FLIP     0x1000
#define VERTICAL_FLIP       0x2000
#define SIZE_8              0x0
#define SIZE_16             0x4000
#define SIZE_32             0x8000
#define SIZE_64             0xC000


//Attribute2 values
#define PRIORITY(n)         ((n) << 10)
#define PALETTE(n)          ((n) << 12)
```

Backgrounds

```
#define REG_BG0CNT              *(u16*)0x4000008
#define REG_BG1CNT              *(u16*)0x400000A
#define REG_BG2CNT              *(u16*)0x400000C
#define REG_BG3CNT              *(u16*)0x400000E
#define REG_BG0HOFS             *(u16*)0x4000010
#define REG_BG0VOFS             *(u16*)0x4000012
#define REG_BG1HOFS             *(u16*)0x4000014
#define REG_BG1VOFS             *(u16*)0x4000016
#define REG_BG2HOFS             *(u16*)0x4000018
#define REG_BG2VOFS             *(u16*)0x400001A
#define REG_BG3HOFS             *(u16*)0x400001C
#define REG_BG3VOFS             *(u16*)0x400001E
#define REG_BG2PA               *(u16*)0x4000020
#define REG_BG2PB               *(u16*)0x4000022
```

```
#define REG_BG2PC              *(u16*)0x4000024
#define REG_BG2PD              *(u16*)0x4000026
#define REG_BG2X               *(u32*)0x4000028
#define REG_BG2X_L             *(u16*)0x4000028
#define REG_BG2X_H             *(u16*)0x400002A
#define REG_BG2Y               *(u32*)0x400002C
#define REG_BG2Y_L             *(u16*)0x400002C
#define REG_BG2Y_H             *(u16*)0x400002E
#define REG_BG3PA              *(u16*)0x4000030
#define REG_BG3PB              *(u16*)0x4000032
#define REG_BG3PC              *(u16*)0x4000034
#define REG_BG3PD              *(u16*)0x4000036
#define REG_BG3X               *(u32*)0x4000038
#define REG_BG3X_L             *(u16*)0x4000038
#define REG_BG3X_H             *(u16*)0x400003A
#define REG_BG3Y               *(u32*)0x400003C
#define REG_BG3Y_L             *(u16*)0x400003C
#define REG_BG3Y_H             *(u16*)0x400003E
#define BG_MOSAIC_ENABLE       0x40
#define BG_COLOR_256           0x80
#define BG_COLOR_16            0x0
#define TEXTBG_SIZE_256x256    0x0
#define TEXTBG_SIZE_256x512    0x8000
#define TEXTBG_SIZE_512x256    0x4000
#define TEXTBG_SIZE_512x512    0xC000
#define ROTBG_SIZE_128x128     0x0
#define ROTBG_SIZE_256x256     0x4000
#define ROTBG_SIZE_512x512     0x8000
#define ROTBG_SIZE_1024x1024   0xC000
#define WRAPAROUND             0x2000
#define CharBaseBlock(n)       (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n)     (((n)*0x800)+0x6000000)
```

Video

```
#define SetMode(mode)    REG_DISPCNT = (mode)
#define VideoBuffer      (u16*)0x6000000
#define BGPaletteMem     (u16*)0x5000000
#define REG_DISPCNT      *(u32*)0x4000000
#define REG_DISPCNT_L    *(u16*)0x4000000
#define REG_DISPCNT_H    *(u16*)0x4000002
#define REG_DISPSTAT     *(u16*)0x4000004
#define REG_VCOUNT       *(u16*)0x4000006
#define REG_WIN0H        *(u16*)0x4000040
#define REG_WIN1H        *(u16*)0x4000042
#define REG_WIN0V        *(u16*)0x4000044
#define REG_WIN1V        *(u16*)0x4000046
#define REG_WININ        *(u16*)0x4000048
#define REG_WINOUT       *(u16*)0x400004A
#define BACKBUFFER       0x10
#define H_BLANK_OAM      0x20
#define OBJ_MAP_2D       0x0
#define OBJ_MAP_1D       0x40
#define FORCE_BLANK      0x80
#define BG0_ENABLE       0x100
#define BG1_ENABLE       0x200
#define BG2_ENABLE       0x400
#define BG3_ENABLE       0x800
#define OBJ_ENABLE       0x1000
#define WIN1_ENABLE      0x2000
#define WIN2_ENABLE      0x4000
#define WINOBJ_ENABLE    0x8000
```

DMA

```
#define DMA_ENABLE               0x80000000
#define DMA_INTERRUPT_ENABLE     0x40000000
#define DMA_TIMING_IMMEDIATE     0x00000000
#define DMA_TIMING_VBLANK        0x10000000
```

```
#define DMA_TIMING_HBLANK            0x20000000
#define DMA_TIMING_SYNC_TO_DISPLAY   0x30000000
#define DMA_TIMING_DSOUND            0x30000000
#define DMA_16                       0x00000000
#define DMA_32                       0x04000000
#define DMA_REPEATE                  0x02000000
#define DMA_SOURCE_INCREMENT         0x00000000
#define DMA_SOURCE_DECREMENT         0x00800000
#define DMA_SOURCE_FIXED             0x01000000
#define DMA_DEST_INCREMENT           0x00000000
#define DMA_DEST_DECREMENT           0x00200000
#define DMA_DEST_FIXED               0x00400000
#define DMA_DEST_RELOAD              0x00600000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE |DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE |DMA_16)
#define REG_DM0SAD      *(u32*)0x40000B0
#define REG_DMA0SAD_L   *(u16*)0x40000B0
#define REG_DMA0SAD_H   *(u16*)0x40000B2
#define REG_DMA0DAD     *(u32*)0x40000B4
#define REG_DMA0DAD_L   *(u16*)0x40000B4
#define REG_DMA0DAD_H   *(u16*)0x40000B6
#define REG_DMA0CNT     *(u32*)0x40000B8
#define REG_DMA0CNT_L   *(u16*)0x40000B8
#define REG_DMA0CNT_H   *(u16*)0x40000BA
#define REG_DMA1SAD     *(u32*)0x40000BC
#define REG_DMA1SAD_L   *(u16*)0x40000BC
#define REG_DMA1SAD_H   *(u16*)0x40000BE
#define REG_DMA1DAD     *(u32*)0x40000C0
#define REG_DMA1DAD_L   *(u16*)0x40000C0
#define REG_DMA1DAD_H   *(u16*)0x40000C2
#define REG_DMA1CNT     *(u32*)0x40000C4
#define REG_DMA1CNT_L   *(u16*)0x40000C4
#define REG_DMA1CNT_H   *(u16*)0x40000C6
#define REG_DMA2SAD     *(u32*)0x40000C8
#define REG_DMA2SAD_L   *(u16*)0x40000C8
```

```
#define REG_DMA2SAD_H    *(u16*)0x40000CA
#define REG_DMA2DAD      *(u32*)0x40000CC
#define REG_DMA2DAD_L    *(u16*)0x40000CC
#define REG_DMA2DAD_H    *(u16*)0x40000CE
#define REG_DMA2CNT      *(u32*)0x40000D0
#define REG_DMA2CNT_L    *(u16*)0x40000D0
#define REG_DMA2CNT_H    *(u16*)0x40000D2
#define REG_DMA3SAD      *(u32*)0x40000D4
#define REG_DMA3SAD_L    *(u16*)0x40000D4
#define REG_DMA3SAD_H    *(u16*)0x40000D6
#define REG_DMA3DAD      *(u32*)0x40000D8
#define REG_DMA3DAD_L    *(u16*)0x40000D8
#define REG_DMA3DAD_H    *(u16*)0x40000DA
#define REG_DMA3CNT      *(u32*)0x40000DC
#define REG_DMA3CNT_L    *(u16*)0x40000DC
#define REG_DMA3CNT_H    *(u16*)0x40000DE
```

Interrupts

```
#define REG_INTERRUPT    *(u32*)0x3007FFC
#define INT_VBLANK        0x0001
#define INT_HBLANK        0x0002
#define INT_VCOUNT        0x0004
#define INT_TIMER0        0x0008
#define INT_TIMER1        0x0010
#define INT_TIMER2        0x0020
#define INT_TIMER3        0x0040
#define INT_COMUNICATION  0x0080
#define INT_DMA0          0x0100
#define INT_DMA1          0x0200
#define INT_DMA2          0x0400
#define INT_DMA3          0x0800
#define INT_KEYBOARD      0x1000
#define INT_CART          0x2000
#define INT_ALL           0x4000
```

Miscellaneous Registers

```
#define REG_MOSAIC        *(u32*)0x400004C

#define REG_MOSAIC_L      *(u32*)0x400004C

#define REG_MOSAIC_H      *(u32*)0x400004E

#define REG_BLDMOD        *(u16*)0x4000050

#define REG_COLEV         *(u16*)0x4000052

#define REG_COLEY         *(u16*)0x4000054

#define REG_SG10          *(u32*)0x4000060

#define REG_SG10_L        *(u16*)0x4000060

#define REG_SG10_H        *(u16*)0x4000062

#define REG_SG11          *(u16*)0x4000064

#define REG_SG20          *(u16*)0x4000068

#define REG_SG21          *(u16*)0x400006C

#define REG_SG30          *(u32*)0x4000070

#define REG_SG30_L        *(u16*)0x4000070

#define REG_SG30_H        *(u16*)0x4000072

#define REG_SG31          *(u16*)0x4000074

#define REG_SG40          *(u16*)0x4000078

#define REG_SG41          *(u16*)0x400007C

#define REG_SGCNT0        *(u32*)0x4000080

#define REG_SGCNT0_L      *(u16*)0x4000080

#define REG_SGCNT0_H      *(u16*)0x4000082

#define REG_SGCNT1        *(u16*)0x4000084

#define REG_SGBIAS        *(u16*)0x4000088

#define REG_SGWR0         *(u32*)0x4000090

#define REG_SGWR0_L       *(u16*)0x4000090

#define REG_SGWR0_H       *(u16*)0x4000092

#define REG_SGWR1         *(u32*)0x4000094

#define REG_SGWR1_L       *(u16*)0x4000094

#define REG_SGWR1_H       *(u16*)0x4000096

#define REG_SGWR2         *(u32*)0x4000098

#define REG_SGWR2_L       *(u16*)0x4000098

#define REG_SGWR2_H       *(u16*)0x400009A

#define REG_SGWR3         *(u32*)0x400009C

#define REG_SGWR3_L       *(u16*)0x400009C
```

```
#define REG_SGWR3_H      *(u16*)0x400009E
#define REG_SGFIFOA      *(u32*)0x40000A0
#define REG_SGFIFOA_L    *(u16*)0x40000A0
#define REG_SGFIFOA_H    *(u16*)0x40000A2
#define REG_SGFIFOB      *(u32*)0x40000A4
#define REG_SGFIFOB_L    *(u16*)0x40000A4
#define REG_SGFIFOB_H    *(u16*)0x40000A6
#define REG_SCD0         *(u16*)0x4000120
#define REG_SCD1         *(u16*)0x4000122
#define REG_SCD2         *(u16*)0x4000124
#define REG_SCD3         *(u16*)0x4000126
#define REG_SCCNT        *(u32*)0x4000128
#define REG_SCCNT_L      *(u16*)0x4000128
#define REG_SCCNT_H      *(u16*)0x400012A
#define REG_P1           *(u16*)0x4000130
#define REG_P1CNT        *(u16*)0x4000132
#define REG_R            *(u16*)0x4000134
#define REG_HS_CTRL      *(u16*)0x4000140
#define REG_JOYRE        *(u32*)0x4000150
#define REG_JOYRE_L      *(u16*)0x4000150
#define REG_JOYRE_H      *(u16*)0x4000152
#define REG_JOYTR        *(u32*)0x4000154
#define REG_JOYTR_L      *(u16*)0x4000154
#define REG_JOYTR_H      *(u16*)0x4000156
#define REG_JSTAT        *(u32*)0x4000158
#define REG_JSTAT_L      *(u16*)0x4000158
#define REG_JSTAT_H      *(u16*)0x400015A
#define REG_IE           *(u16*)0x4000200
#define REG_IF           *(u16*)0x4000202
#define REG_WSCNT        *(u16*)0x4000204
#define REG_IME          *(u16*)0x4000208
#define REG_PAUSE        *(u16*)0x4000300
```

Timers

```
#define REG_TM0D        *(u16*)0x4000100
#define REG_TM0CNT      *(u16*)0x4000102
#define REG_TM1D        *(u16*)0x4000104
#define REG_TM1CNT      *(u16*)0x4000106
#define REG_TM2D        *(u16*)0x4000108
#define REG_TM2CNT      *(u16*)0x400010A
#define REG_TM3D        *(u16*)0x400010C
#define REG_TM3CNT      *(u16*)0x400010E
#define FREQUENCY_0     0
#define FREQUENCY_64    1
#define FREQUENCY_256   2
#define FREQUENCY_1024  1 | 2
#define TIMER_CASCADE   4
#define TIMER_IRQ       64
#define TIMER_ENABLE    128
```

Appendix D

Answers to the Chapter Quizzes

This appendix contains the answers to all the quiz questions from each chapter. I hope you got all the answers correct! If you miss more than three answers to any given quiz, I recommend that you go back and reread the relevant chapter and try again before proceeding. Good luck!

Chapter 1

1. ■	6. A
2. C	7. C
3. A	8. D
4. D	9. A
5. B	10. B

Chapter 2

1. A	6. C
2. C	7. D
3. D	8. A
4. C	9. B
5. B	10. C

Chapter 3

1. A	6. B
2. B	7. C
3. C	8. C
4. B	9. A
5. D	10. D

Chapter 4

1. C	6. B
2. B	7. B
3. A	8. C
4. A	9. D
5. D	10. A

Chapter 5

1. B	6. B
2. A	7. A
3. B	8. C
4. C	9. D
5. D	10. D

Chapter 6

1. A	6. A
2. C	7. C
3. B	8. D
4. D	9. B
5. B	10. A

Chapter 7

1. B	6. B
2. C	7. B
3. D	8. C
4. A	9. A
5. D	10. A

Chapter 8

1. C	6. B
2. A	7. B
3. B	8. A
4. D	9. D
5. A	10. C

Chapter 9

1. C	6. D
2. A	7. C
3. C	8. B
4. B	9. A
5. A	10. D

Chapter 10

1. C	6. C
2. A	7. B
3. D	8. D
4. A	9. A
5. C	10. B

Chapter 11

1. A	6. B
2. B	7. D
3. A	8. A
4. C	9. B
5. D	10. B

Appendix E

Using The CD-ROM

The CD that comes with this book contains some important files that you will want to use when working through the sample programs in the book. The most important files on the CD are the source code files for the sample programs in the book.

The programs are stored in folders on the CD that are organized by chapter from the root \Sources folder. Inside \Sources, you will find chapter sub-folders: \Sources\Chapter01, \Sources\Chapter02, and so on. I recommend that you copy the entire \Sources folder to your hard drive, turn off the read-only property for all of the files, so you will be able to peruse the sample projects for the book more easily.

This book is about writing Game Boy Advance programs with a GCC compiler chain distribution called HAM, so I have included a version of HAM on the CD that you can install and use while reading the book and typing in sample programs. HAM is free for both personal and professional use, and is based on an open-source C/C++ compiler and ARM assembler, collectively known as a "compiler chain."

Everything you need to write Game Boy Advance programs is installed with HAM, including the emulator. There is also a \Tools folder with all of the various utility programs and other software used in the book, such as gfx2gba and VisualBoyAdvance.

Develop your own games for the Nintendo® Game Boy® Advance!

Now you can write, compile, and run Game Boy programs right on your Windows® desktop! *Programming the Nintendo Game Boy Advance: The Unofficial Guide* will show you how you can use the free HAM SDK to create your own games for the Game Boy Advance. When you're ready to take the game off of your desktop and run it on a real Game Boy, you'll get plenty of coverage on your options with multi-link cables and flash linkers as well as details for purchasing either. Staying focused as a true console programming book, *Programming the Nintendo Game Boy Advance: The Unofficial Guide* gives you the tips and skills that you need to develop a career as a console game programmer. All you need to get started is experience programming with C and professional game programming aspirations.

Jonathan S. Harbour has been programming games for 15 years, beginning with Microsoft GW-BASIC and Turbo Pascal, then moving on to Turbo C, Borland C++, Watcom C++, and 80386 assembler. After bridging the gap to Windows programming, he has spent time with Borland Delphi, Visual Basic, Visual C++, and Visual Studio .NET. Jonathan graduated from DeVry Institute of Technology in 1997 with a B.S. in Computer Information Systems.

Series Editor

André LaMothe, CEO Xtreme Games LLC

André LaMothe has been involved in the computing industry for more than 25 years. His experience includes 2D/3D graphics, AI research at NASA, compiler design, robotics, virtual reality, and telecommunications. His books are top sellers in the game programming genre, and his experience and guidance are echoed in all Premier Press *Game Development* books.
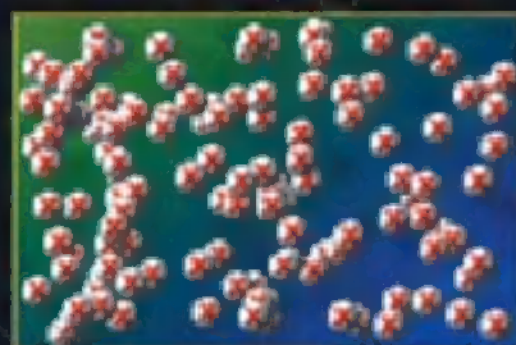
MODE 4
240x160

WHAT'S ON THE CD

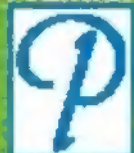- COMPLETE SOURCE CODE FOR EACH CHAPTER PROJECT
- VISUAL HAM INTEGRATED DEVELOPMENT ENVIRONMENT
- HAM DEVELOPMENT KIT FOR COMPILING GBA PROGRAMS
- VISUAL BOY ADVANCE GBA EMULATOR FOR WINDOWS 98/ME/2000/XP
- PRE-COMPILED ROMS READY TO RUN ON YOUR GBA USING A LINK CABLE